ARM Target Development System

User Guide

Document number: ARM DUI 0061A
Issued: March 1997
Copyright ARM Ltd. 1997

ENGLAND
ARM
90 Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@arm.com

GERMANY
ARM
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@arm.com

JAPAN
ARM
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@arm.com

USA
ARM
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

Open Access

Proprietary Notice

ARM, the ARM Powered logo and EmbeddedICE are trademarks of ARM Ltd.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

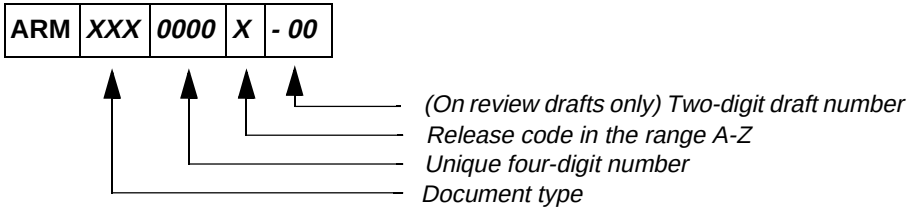
The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties or merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Ltd shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.



Document Status

The document's status is displayed in a banner at the bottom of each page. This describes the document's confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above, and to staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A	March 1997	HAT	First issue

Open Access

Contents

1 About This Document 1-1
- 1.1 Overview 1-2
- 1.2 Feedback 1-3

2 Setting up your Target System 2-1
- 2.1 Overview 2-2
- 2.2 Choosing a Debugging System 2-4
- 2.3 Setting up the ARM Development Board 2-11
- 2.4 Communicating with EmbeddedICE 2-13
- 2.5 Communicating with Angel 2-20
- 2.6 Communicating with Angel using EmbeddedICE 2-29
- 2.7 Using the Flash Downloader 2-31

3 A Worked Example 3-1
- 3.1 Getting Started with the Example Code 3-2
- 3.2 Using the Code with the Windows Toolkit 3-3
- 3.3 Using the Code on the Command Line 3-7

4 Test Equipment 4-1
- 4.1 Introduction to Real-Time Trace 4-2
- 4.2 The Test Interface 4-3

Open Access

Contents

5 Programmer's Model of the ARM Development Board 5-1

5.1 Introduction 5-2
5.2 Development Board Architecture 5-3
5.3 Main Memory Map 5-7
5.4 RAM Region 5-9
5.5 ROM Region 5-12
5.6 Reference Peripherals 5-13
5.7 Serial, Parallel, and PC Card Ports 5-24

6 Configuring the ARM Development Board 6-1

6.1 Introduction 6-2
6.2 Setting the Main System Clock Frequency 6-4
6.3 Remapping the Memory 6-5
6.4 Setting the SRAM Speed and Width 6-6
6.5 Setting the EPROM/Flash Speed and Width 6-7
6.6 Parallel Port/LED mode configuration 6-9
6.7 Big-endian Operation 6-12

7 Example Code Tutorials 7-1

7.1 The Example Code Suite 7-2
7.2 Driving the Parallel Port 7-4
7.3 Driving the Serial Port 7-6
7.4 A Simple Interrupt Handler Routine 7-9
7.5 Simple PC Card (PCMCIA) Access 7-12
7.6 PC Card (PCMCIA) SRAM Memory Card Read/Write Example 7-14
7.7 A ROM Resident Program 7-18
7.8 Flash Downloader 7-24

8 Benchmarking on the ARM Development Board 8-1

8.1 Dhrystone Benchmarking Test Function 8-2
8.2 Full Results 8-10

9 ARM Development Board Diagnostic Utility 9-1

9.1 Introduction 9-2
9.2 selftest 9-3
9.3 diagnose 9-4

A ST16C552 UART with FIFO and Parallel Printer Port A-1

A.1 Description A-2
A.2 Serial Port Programming A-3
A.3 Printer Port Programming A-15

Glossary Glossary-1

Index Index-1

Open Access

1 About This Document

1.1	Overview	1-2
1.2	Feedback	1-3

Open Access

About This Document

1.1 Overview

This user guide describes how you set up your target development system.

1.1.1 Typographical Conventions

The following typographical conventions are used in this document:

`typewriter`	text that may be entered at the keyboard; commands, file and program names, and C source
`typewriter-italic`	text which must be submitted with user-supplied information; this is most often used in syntax descriptions
italic	important notes and ARM-specific terminology
bold	signal names and menu selections
bold-italic	internal cross-references

Filenames

Unless otherwise stated, filenames are quoted in MS-DOS format, for example:

```
EXAMPLES\BASICASM\GCD1.S
```

If you are using the Unix platform, you must translate them into their Unix equivalent:

```
examples/basicasm/gcd1.s
```


Open Access

1.2 Feedback

1.2.1 Feedback on this manual

If you have any comments or suggestions about this document, please contact your supplier giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- a concise explanation of your comments

1.2.2 Feedback on the Target Development System

If you have any problems with the Target Development System, please contact your supplier including:

- details of the setup you are using
- details of the versions of hardware and software you are using
- a clear explanation of what you expected to happen, and what actually happened

Open Access

About This Document

Open Access

2 Setting up your Target System

2.1	Overview	2-2
2.2	Choosing a Debugging System	2-4
2.3	Setting up the ARM Development Board	2-11
2.4	Communicating with EmbeddedICE	2-13
2.5	Communicating with Angel	2-20
2.6	Communicating with Angel using EmbeddedICE	2-29
2.7	Using the Flash Downloader	2-31

Open Access

Setting up your Target System

2.1 Overview

This chapter describes how you set up your target development system. To do this, you must first choose the type of debugging system you are going to use, and select the means of communication between the host computer and the target system.

Next, follow through the relevant sections in this chapter to complete your setup. A simple worked example is provided in ***Chapter 3, A Worked Example*** for you to confirm that your setup is correct.

The ARM Target Development System comprises:

- a target board, such as the ARM Development Board
- a host computer running a debugger
- a debugging system

2.1.1 The target development board

The target development board is a platform suitable for real-time code development and exploration of embedded ARM processors. It is a convenient means of evaluating the Advanced RISC Machines' family of RISC processors.

Later in the development cycle, the target board may be a development version of the product hardware.

This guide focuses on the ARM Development Board (7TDMI version) as the development platform.

Other development boards are available from ARM and other third parties. For the latest status, see Development Support at www.arm.com. If you are using a board other than the 7TDMI version, refer to the documentation supplied with the board for information about how to configure it.

2.1.2 The host computer

This user guide assumes that your host computer is running the debugger in the ARM Software Development Toolkit, version 2.1x.

If you are using an earlier version of the Software Development Toolkit, you are strongly recommended to upgrade to the most recent version. Otherwise, consult the documentation supplied with your version for information about how to configure the debugger and use the Demon Debug Monitor.

Other software toolkits for the ARM, available from third parties, are not discussed in this guide. If you are using a third-party toolkit, use this guide alongside the documentation supplied with the toolkit.

Open Access

2.1.3 The debugging system

ARM offers a choice of debugging systems for your target development system:

- the ARMulator
- EmbeddedICE
- Angel Debug Monitor

The characteristics and advantages of each method are discussed in ***2.2 Choosing a Debugging System*** to help you to decide which system best suits your needs.

If you have already chosen a debugging system, you can skip ***2.2 Choosing a Debugging System***.

Open Access

2.2 Choosing a Debugging System

There are three debugging systems available for software applications developed to run on an ARM core:

- the ARMulator
- EmbeddedICE
- Angel Debug Monitor (with or without EmbeddedICE)

These systems are described in the following three sections.

2.2.1 The ARMulator

The ARMulator is a software emulator of the ARM processor. It is a simulator core that emulates the instruction sets of various ARM processors.

The ARMulator enables you to develop your software before any hardware is available. It does this by providing an environment for the development of ARM-targeted software on non ARM-based host systems. You do not need a target system to use the ARMulator.

The ARMulator is included in the ARM Software Development Toolkit.

2.2.2 EmbeddedICE

EmbeddedICE is a JTAG-based, non-intrusive, debugging system for ARM processors. EmbeddedICE provides the interface between a debugger and an ARM core either on a development board, or embedded within any *Application Specific Integrated Circuit (ASIC)*, using the JTAG port for communication.

EmbeddedICE provides you with:

- real-time address and data-dependent breakpoints (including ROM)
- single stepping
- full access and control of the ARM core
- full memory access (read and write)
- full I/O system access (read and write)

EmbeddedICE also enables the embedded microprocessor to access host system peripherals, such as screen display, keyboard input, and disk drive storage.

EmbeddedICE is an extension to the architecture of the ARM family of RISC processors, and provides the ability to debug cores that have been deeply embedded into systems. It consists of four parts:

- a set of debug extensions to the ARM core
- the *EmbeddedICE macrocell*, to provide access to the extensions from the outside world
- the EmbeddedICE interface, to provide communication between the EmbeddedICE macrocell and the debugger resident on the host computer

Open Access

- the Debug Comms Channel, an additional communication channel over the JTAG interface between the target ARM processor and host computer

Debug extensions to the ARM core

The debug extensions consist of a number of scan chains around the core and some additional pins that are used to control the behavior of the core for debug purposes.

The three most significant pins are:

BREAKPT — This enables external hardware to halt execution of the processor for debug purposes. When HIGH, the current memory access is tagged as breakpointed. If the memory access is an instruction fetch, the core will enter debug state if and when the instruction reaches the execute stage of the pipeline. If the memory access is for data, the core will enter debug state when the current instruction completes execution. The EmbeddedICE macrocell has control of this input to the core.

DBGRQ — This is level-sensitive input that causes the core to enter debug state immediately the current instruction has completed execution. This enables external hardware to force the ARM into debug state.

DBGACK — This is an output from the ARM that goes HIGH when the core is in debug state. This enables other peripherals or the debugging system to determine the current state of the core, and act accordingly.

Refer to the ***Debug Interface*** section of the appropriate ARM data sheet for further details.

EmbeddedICE macrocell

The EmbeddedICE macrocell is the integrated on-chip logic that provides debug support for ARM cores.

The EmbeddedICE macrocell is programmed in serial through the *Test Access Port (TAP)* controller on the ARM, via the JTAG interface. This is usually achieved via the EmbeddedICE interface.

The EmbeddedICE macrocell consists of two real-time watchpoint units, together with a control and status register. One or both watchpoint units can be programmed to halt the execution of instructions by the ARM core using its **BREAKPT** signal. Execution is halted when a match occurs between the values programmed into the EmbeddedICE macrocell and the values currently appearing on the address bus, data bus and various control signals. Any bit can be masked to prevent it affecting the comparison. Either watchpoint unit can be configured to be a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches).

Refer to the ***EmbeddedICE macrocell*** (or ***ICEBreaker***) section of the appropriate ARM data sheet for further details.

EmbeddedICE interface

The EmbeddedICE interface is a JTAG protocol conversion unit. This translates the debug protocol messages generated by the debugger into JTAG signals that can be sent to the EmbeddedICE macrocell, and vice versa. The interface can be connected to the host

Open Access

computer using either the built-in serial port, or the built-in serial and parallel ports. The serial port is used for bidirectional transfers, but the parallel port is only used to download code. Using the parallel port increases the maximum download speed with EmbeddedICE to approximately 20KB/s. This is considerably greater than using the serial ports alone (even at 38400 baud), and so is the recommended method of using EmbeddedICE (host permitting).

See ***2.4 Communicating with EmbeddedICE*** on page 2-13 for information about setting up the EmbeddedICE system.

The Debug Comms Channel

In addition to the debugger's communication with the processor via the scan chains, a second communication channel exists using Coprocessor 0 instructions to access the EmbeddedICE macrocell registers. This is called the Debug Comms Channel.

The *Angel Debug Protocol (ADP)* compatible EmbeddedICE interface has two channels that deal with data on the Debug Comms Channel. This makes it cleaner to process at both ends. It is also considerably easier to make use of the Debug Comms Channel from the host end, because the standard handler, which reads and writes data to or from a file, can be replaced.

On the target, you must provide code to drive the Debug Comms Channel directly using coprocessor instructions. The Angel Debug Monitor is not required to be running on the target.

If you are using the *ARM Debugger for Windows (ADW)*, you can open a channel viewer window that enables direct input and output via the Debug Comms Channel.

On a UNIX system under armsd, you can use the `ccin` and `ccout` commands to direct output from the Debug Comms Channel to a file or take input for the Debug Comms Channel from a file.

The effect of EmbeddedICE on the debuggee

Although EmbeddedICE is generally non-intrusive, there are two exceptions:

- When an ARM debugger is started, it attempts to find out the state of the debuggee. To do this, it halts the debuggee and inspects the state of the ARM registers. This, however, can be considered non-intrusive if the debugging session begins after the debugger has been started.
- Watchpoints on structures or arrays that are larger than one word may cause the debuggee to halt execution when writes occur close to the watchpointed area. EmbeddedICE will restart execution transparently to the user, but this may still cause problems if the application is real-time.

Open Access

2.2.3 The Angel Debug Monitor

To aid readability, the term *Angel* is used to mean *the Angel Debug Monitor* throughout this user guide.

Angel is a program that enables rapid development and debugging of applications running on ARM-based hardware. Angel runs on either a development board for ARM processors, or a development version of the product hardware alongside your application. It communicates with a debugger that can handle the Angel communications protocol, such as the ARM Debugger for Windows, or armsd.

If you use Angel for debugging using one of ARM's development or evaluation boards, you can simply use the Angel software supplied on the release CD or in the Flash on the board. If you intend to use Angel to debug your application on your own custom hardware, you must port Angel to run on this hardware.

Angel can debug applications running in both ARM and Thumb state on target hardware.

A typical Angel system has two main components that communicate via a physical link, such as a serial cable:

Debugger	This runs on the host computer. It gives instructions to Angel and displays the results obtained from it. The debugger could be armsd, the ARM Debugger for Windows, or any other debugging tool that can handle the communications protocol used by Angel.
Angel	This runs alongside the application being debugged on the target platform. There are two versions of Angel: • a full version for use on development hardware • a minimal version for use on production hardware

Angel uses a debugging protocol called the Angel Debug Protocol (ADP), which supports multiple channels. It is easy to port to different hardware. It requires control over the ARM's exception vectors, but can share these with your applications.

You can use Angel to:

- evaluate existing application software on real hardware (as opposed to emulation of the hardware)
- develop software applications on development hardware
- bring into operation new hardware that includes an ARM processor
- port operating systems to ARM-based hardware

These activities require some understanding of how Angel's components work together. Some involve modification of Angel itself, such as porting operating systems.

This chapter provides an overview of Angel. For more detailed information, see the *ARM Software Development Toolkit User Guide* (ARM DUI 0040) and the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

Open Access

Angel and Demon

Demon is the debug monitor supplied with the ARM Software Development Toolkit release 2.0x. Angel is similar in functionality to Demon. However, Angel is more modular in design, easier to port, and easier to customize. It also supports a number of facilities that are unavailable with Demon.

For more information about the differences between Angel and Demon, see the *ARM Software Development Toolkit User Guide* (ARM DUI 0040).

Angel on the ARM Development Board

Angel on the ARM Development Board can communicate with the host computer via serial, parallel, or Ethernet channels. Ethernet connection is via the PC Card interface for which an Ethernet adaptor is supplied as an extra. This increases the maximum download speed to approximately 100KB/s.

See ***2.5 Communicating with Angel*** on page 2-20 for information about setting up with Angel.

2.2.4 How EmbeddedICE differs from a debug monitor

A debug monitor, such as Angel, is an application that runs on the board in conjunction with the user application, and requires some resource to be available for it on the board. When the board powers up, Angel installs itself by initializing the vector table so that it takes control of the board when an exception occurs. Communication coming in from the host causes an interrupt, halting the user application and calling the appropriate code within Angel. Angel then returns to the user application. This can complicate matters if the application also requires access to interrupts. Similarly, if the application requests some form of I/O to the host, this is implemented within the application using a *software interrupt (SWI)* instruction that is dealt with by Angel's SWI handler. This means that Angel requires ROM to store the debug monitor code, RAM to store its data, and control over the exception vectors to enable it to gain control of the ARM while the user's application is running.

EmbeddedICE, on the other hand, requires no such resource. Rather than existing as an application on the board, it works by using a combination of the debug hardware on the core and the interface that handles communication between the core's debug hardware and the host. EmbeddedICE has been designed so that debugging via the JTAG port is as non-intrusive as possible:

- The debuggee does not require special hardware to support debugging (the debug extensions and the EmbeddedICE macrocell are all that is necessary).
- The debuggee system does not require memory to be set aside for debugging, or special software to be incorporated to allow debugging.
- Execution of the debuggee is halted only when a breakpoint or watchpoint has been hit or the user requests that the debuggee is halted.

Note 1 *Although EmbeddedICE requires no memory on the target to operate, the target still requires some memory to execute the application code.*

Open Access

Note 2 *EmbeddedICE may pollute the exception vectors if semihosting and/or vector catch are enabled.*

EmbeddedICE interacts with the target processor by halting execution and then forcing the processor into debug state to access data required by the debugger via the JTAG interface. This results in the processor being halted for significant periods of time, which may cause problems when you are debugging real-time systems.

Angel is invoked by SWI instruction and does not halt the processor. Instead, control is passed from the application code to the debug monitor. This may reduce the amount of dead time when data is accessed on behalf of the debugger. Furthermore, interrupts can be disabled for the minimum time needed by the debugger, allowing real-time application interrupts to be serviced.

Angel can be used on the final product even if no extra communication ports are available by using the Debug Comms Channel with EmbeddedICE. See section ***2.6 Communicating with Angel using EmbeddedICE*** on page 2-29.

See ***Table 2-1: Summary of differences between Angel and EmbeddedICE*** for a summary.

Open Access

	Angel	EmbeddedICE
Run control	Does not halt the processor. This may be important when debugging real-time systems.	Breakpoints available in ROM. Hardware watchpoints at full processor speed.
System resources	Requires some target system ROM and RAM, and a software interrupt.	Requires no system resources.
Host communication	Can use serial, serial and parallel, or Ethernet communication ports, if available on the target system. Can use the Debug Comms Channel with the EmbeddedICE interface if no other communications port is available.	Only requires the JTAG interface to the ARM processor.
Download speed	Ethernet option can enable 100 KB/s.	Current EmbeddedICE Interface Unit is limited to approximately 20KB/s.
Pre-requisites	Angel must be ported to the target system. Angel is ported to all ARM Development Boards	ARM processor must have the debug extensions (on-chip). Can be used on a target with untested memory and peripherals systems, because communication is directly with the ARM processor.

Table 2-1: Summary of differences between Angel and EmbeddedICE

Open Access

2.3 Setting up the ARM Development Board

2.3.1 Electrostatic warning

The ARM Development Board is shipped in protective packaging.

Take care not to subject the board to high electrostatic potentials. It is advisable to wear a grounding strap or similar protective device when you handle the board.

Avoid touching the component pins or any other metallic element.

2.3.2 Requirements

To set up the ARM Development Board you need:

- the host computer running the ARM Software Development Toolkit
- the ARM Development Board
- suitable connection between the host and the development board (see the following sections)
- DC power supply with the following outputs:

  +5V @ 3A

  +12V @ 500mA

  A suitable source is either a bench power supply or a PC-type power supply. A connector on the board is provided to enable a standard PC supply to be connected.

Note *The +12V is only required for the PC Card slots (for example, for the Ethernet adaptor).*

2.3.3 Powering up the board

It is essential that the power lines are properly connected to the board. Failure to observe this may lead to damage to the board.

Figure 2-1: Power connector illustrates how the power connections must be made. The locations of the power connectors (P8 and P9) on the board are illustrated in ***Figure 2-2: Connecting up the development board with EmbeddedICE*** on page 2-13.

Open Access

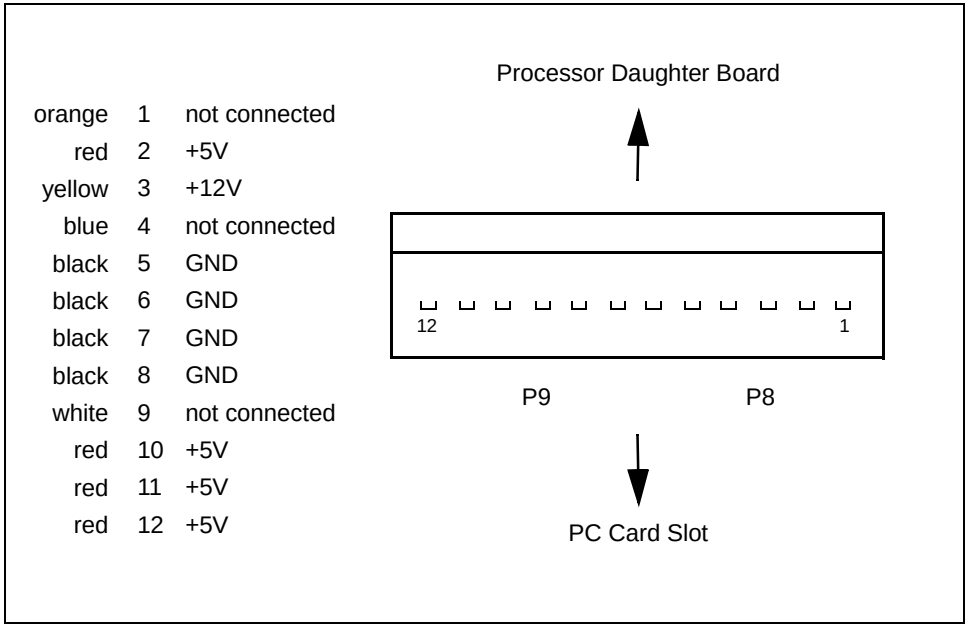


Figure 2-1: Power connector

When you switch the power on, the two green LEDs marked “+5V” and “+3V3” (adjacent to the power connector) light. If they do not, switch off and check the supply polarity.

The green LED marked “FPGA OK” also lights to indicate that the *Advanced Peripheral Bus (APB) field programmable gate array (FPGA)* has initialized correctly. This takes only a fraction of a second. If it does not light, refer to the hardware reference guide supplied with the development board. The six amber LEDs may also light.

Note *The LED marked “FPGA OK” may not light if you are trying to reprogram the FPGA with different functionality.*

To continue setting up your target system, power down the development board, and proceed to ***2.4 Communicating with EmbeddedICE*** on page 2-13 if you are using EmbeddedICE, or ***2.5 Communicating with Angel*** on page 2-20 if you are using Angel.

Open Access

2.4 Communicating with EmbeddedICE

You will need the following:

- ARM Development Board
- EmbeddedICE interface unit
- 9-pin RS232 cable
- 14-way ribbon cable
- 25-pin parallel cable (optional)
- DC power supply, 7-9V @ 500mA (not supplied)
- DC power supply, 5V @ 3A, 12V @ 500mA (not supplied)

2.4.1 Connecting the EmbeddedICE unit

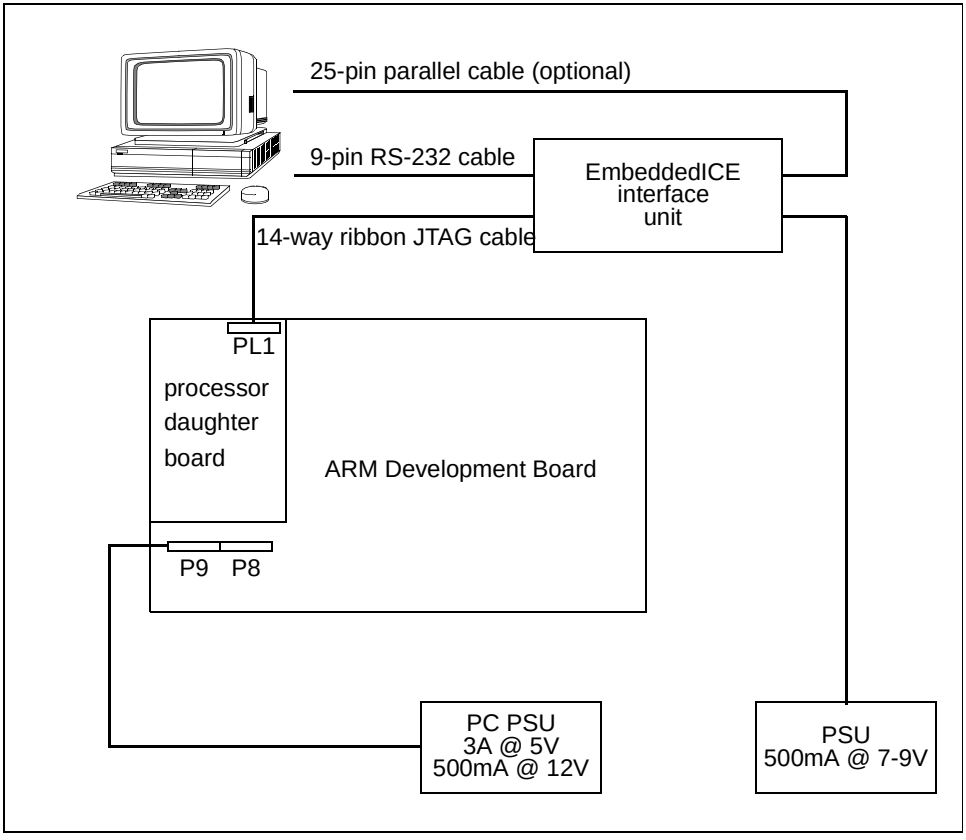


Figure 2-2: Connecting up the development board with EmbeddedICE

Open Access

To communicate between the host computer and the target system using the EmbeddedICE interface:

1 Connect the serial cable between the 9-pin serial port on the EmbeddedICE interface and the serial comms port of the host PC.

2 Connect the parallel port cable between the 25-pin parallel port connector on the EmbeddedICE interface and the printer port on the host PC. (This is not essential, but will speed up download.)

3 Connect the EmbeddedICE interface to an external +7 to +9V DC (unregulated) power supply, 500mA or greater (not supplied).

Note that the interface has diode protection to prevent reversed supplies from damaging the board. For correct operation, the 2.1mm power connector should have the positive supply connected to the center pin, as shown in ***Figure 2-3: Power connection***:

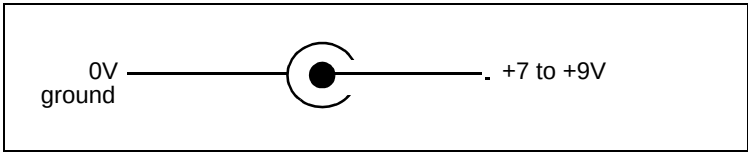


Figure 2-3: Power connection

4 Ensure that the REMAP link (LK18) on the development board is in the “out” position (the two pins of the link should not be connected together).

5 Power up the EmbeddedICE interface and press the red reset button.

On releasing the reset button, the LED on the unit lights up. If the LED immediately blinks off once for a fraction of a second, the software version is *Remote Debug Protocol (RDP)* compatible, not *Angel Debug Protocol (ADP)* compatible.

6 Power up the development board. See ***2.3.3 Powering up the board*** on page 2-11.

7 Connect the EmbeddedICE interface unit to the target development board using a 14-way ribbon JTAG cable. (The actual location of the JTAG connection on the target board depends on the board you are using—it is on the top edge (PL1) of most processor daughter boards.)

Open Access

2.4.2 Configuring the ARM Debugger for Windows to use EmbeddedICE

Configuring the ARM Debugger for Windows

To configure the *ARM Debugger for Windows* (*ADW*) to access a remote target rather than the ARMulator:

1. Start the ADW (not via the ARM Project Manager). In its default state, a message similar to the following is displayed in the console window:

```
ARMULATOR 2.0 [Jan 6 1997]ARM7DTMI, 15.0MHz, 4Gb, Dummy
  MMU, Soft Angel 1.4,[Angel SWIs, Demon SWIs],FPE,
  Profiler, Tracer, Little endian
```

2. Choose **Configure Debugger...** from the Options menu. The Debugger Configuration property sheet is displayed.
3. Click on the **Target** tab.

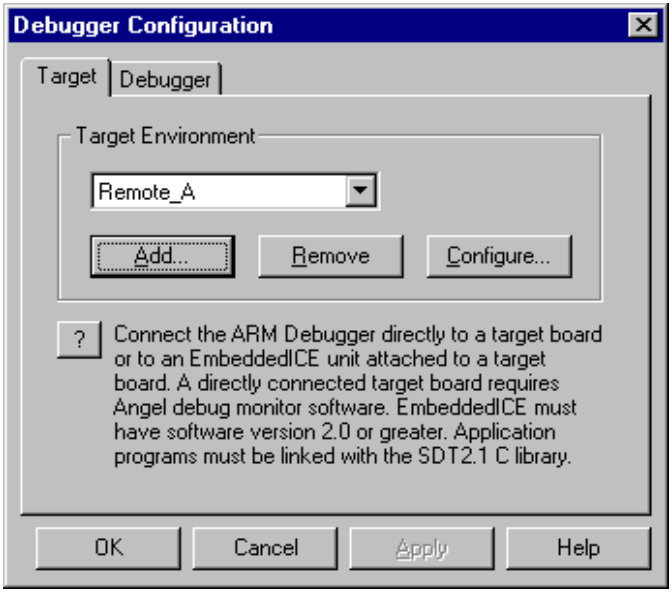

Debugger Configuration
Target
Debugger
Target Environment
Remote_A
Add...
Remove
Configure...
?
Connect the ARM Debugger directly to a target board or to an EmbeddedICE unit attached to a target board. A directly connected target board requires Angel debug monitor software. EmbeddedICE must have software version 2.0 or greater. Application programs must be linked with the SDT2.1 C library.
OK
Cancel
Apply
Help

Figure 2-4: The Debugger Configuration property sheet

4. Select **Remote_A** for ADP-compatible EmbeddedICE units.

   Note *If you are using an older RDP version of the EmbeddedICE unit (the LED blinks on-off-on at reset),* **Add** *the configuration file* `Arm210\Bin\remote_d.dll` *for the Target Environment, and select* **Remote_D**. *Alternatively, upgrade your EmbeddedICE ROM (see http://www.arm.com for the latest version).*

Open Access

5 Click on **Configure...**

6 If you are using both serial and parallel cables, select **Serial/Parallel** from **Remote Connection**. If you are using only a serial cable, select **Serial**.

7 Select the serial and parallel communication ports, and the serial line speed. (The ARM Development Board operates at 38400 bps.)

8 Click on **OK**.

9 If the target board being used runs in big-endian mode (the default is little-endian mode), click on the **Debugger** tab, and click on **Big** in the Endian selection.

10 Click on **OK**.

The debugger is restarted. The Restarting dialog box is displayed. Rapidly changing numbers are shown, indicating reading and writing to the target.

Note *If the target board does not respond (the numbers in the Restarting dialog box do not change), ADP times out within approximately five seconds and defaults back to ARMulator.*

When the configuration has completed successfully, a message similar to the following is displayed in your host console window:

```
EmbeddedICE Manager (ADP ARM7TDI) 2.01 (Advanced RISC Machines)
```

Configuring EmbeddedICE interface to match the target core

The EmbeddedICE interface unit must be configured for the ARM core present in the target system. Typically ARM cores are:

ARM7DI	ARM7 core with debug extensions and EmbeddedICE macrocell (includes ARM7DMI)
ARM7TDI	ARM7 core with Thumb, debug extensions, and EmbeddedICE macrocell (includes ARM7TDMI)

When you start EmbeddedICE, it automatically defaults to a particular configuration according to which version of the interface software is installed:

- versions up to 1.03 default to ARM7DI
- versions after this default to ARM7TDI

If any of the following messages are displayed when you start up the debugger (or reload an image), you must reconfigure the EmbeddedICE:

```
Error during initialization:
Recoverable error in RDI initialization
Target processor not stopped
```

Note *You should ensure that the selected configuration is correct, otherwise you may have problems when you run the downloaded program.*

With a target ARM7DI, using the ARM7TDI configuration can lead to breakpoints not being hit. If you press ^C (using UNIX armsd) or the stop button (using ADW) to interrupt the execution, you enter the undefined instruction trap.

Open Access

To configure the EmbeddedICE interface

1 Choose **Configure EmbeddedICE** from the Options menu. The EmbeddedICE Configuration dialog box is displayed:

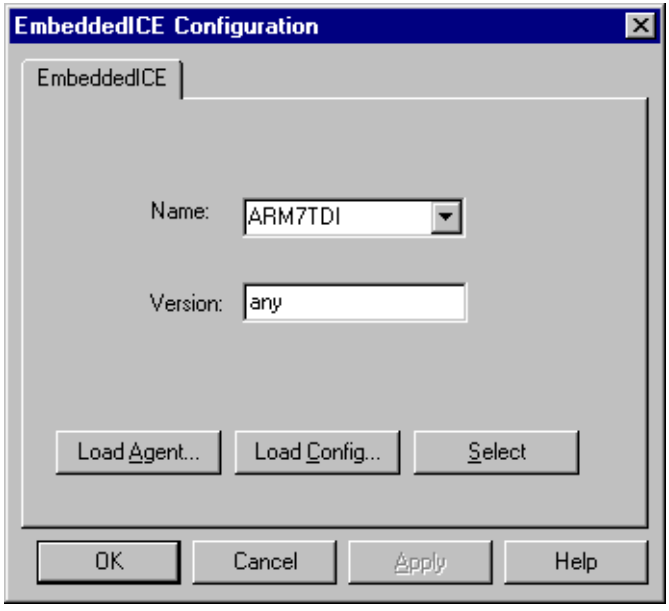


Figure 2-5: The EmbeddedICE Configuration dialog box

2 Do one of the following:

- Choose the name from the list and type in the version. You can usually type `any` for the version. Click on **Select**.
- Load a new agent: click on **Load Agent...**, select the required agent file (usually found in `ARM210\Angel\Images\iceagent`), and then click on **Open**.
- Load a new configuration file: click on **Load Config...**, select the required configuration file, and then click on **Open**. The required configuration data is usually contained within the agent itself, so you seldom need to use this option.

3 Click on **OK**.

When the debugger closes down, it remembers the remote debugging option settings, so you can automatically use them again next time you use the debugger.

Open Access

2.4.3 Configuring the armsd debugger to use EmbeddedICE

Configuring armsd

When you initialize armsd using EmbeddedICE, it is necessary to tell armsd the endianness of the target. This contrasts with running Angel, or using ARMulator, where there is a default endianness (little-endian).

To configure for a little-endian target using EmbeddedICE version 2.0 or later, the standard serial and parallel ports, and the default linespeed (9600 bps), enter:

```
armsd -li -adp -port s,p
```

For a big-endian target using serial port 2 (rather than serial port 1), and linespeed of 38400, enter:

```
armsd -bi -adp -port s=2,p -linespeed 38400
```

where:

`-bi` is the big-endian switch. Use `-li` for little-endian.

`-adp` is the switch to allow communication using the ARM Debug Protocol required by Angel.

`-port s=2,p` indicates that the serial port is port 2 on the host computer, and the parallel port is the standard port.

The serial and parallel ports can be shown numerically (1 or 2) or be the host device name required.

`-linespeed 38400` sets the baud rate for serial communications between the host computer and the Development System.

If you are using a version of EmbeddedICE earlier than 2.00 (that uses RDP rather than ADP communication), enter:

```
armsd -li -serpar
```

or for a big-endian target, serial port 2 (rather than serial port 1), and linespeed of 38400, enter:

```
armsd -bi -serpar -port 2 -linespeed 38400
```

When you enter the correct command, the host displays the debugger's opening banner, similar to:

```
A.R.M. Source-level Debugger vsn 4.46 (Advanced RISC Machines
   SDT 2.10) [Dec13 1996]
EmbeddedICE Manager (ADP ARM7TDI) 2.01 (Advanced RISC Machines)
```

Configuring the EmbeddedICE interface to match the target core

The following commands enable you to access EmbeddedICE configuration and agent facilities from within armsd:

`listconfig` Lists the configurations known to the debug agent. Note that the first configuration listed is the default for the version of the agent in use.

Open Access

`selectconfig` *`name version`*

Specifies the target configuration to use.

`name` indicates the configuration to use, typically `ARM7DI` or `ARM7TDI`.

`version` indicates the version to be used:

- `any` accept any version (default)
- *n* must use version *n*
- *n*+ must use version *n* or later

The highest numbered version meeting the version constraint is used.

`loadagent file` Downloads the replacement agent software into EmbeddedICE and starts it in RAM, rather than using the one in ROM. See the *ARM Software Development Toolkit User Guide* (ARM DUI 0040) for further details.

`loadconfig file` Specifies the file containing configuration data to be loaded. Note that this option is seldom needed, because the required configuration data is usually contained within the agent itself.

For further information on using EmbeddedICE, see the *ARM Software Development Toolkit User Guide* (ARM DUI 0040).

Open Access

2.5 Communicating with Angel

Note *If you are developing systems using Angel, you must take into account additional considerations. These are documented in Application Note 43:* Design Considerations when using Angel Debug Monitors *(ARM DAI 0043).*

2.5.1 Setting up the ARM Development Board

Your development board must have Angel resident in its Flash or EPROM:

8-bit EPROM/Flash U12 on the ARM Development Board

16-bit EPROM/Flash U13 on the ARM Development Board

You can obtain the latest version for the ARM Development Board from http://www.arm.com.

If your development board has no image loaded into its Flash or EPROM, you must connect to the board using EmbeddedICE and download Angel following the instructions in ***2.7 Using the Flash Downloader*** on page 2-31. When you have downloaded, exit the debugger, and power down.

When Angel is loaded onto your development board, you can communicate via serial, parallel, Ethernet, or EmbeddedICE (see ***2.6 Communicating with Angel using EmbeddedICE*** on page 2-29). Ethernet requires an additional PC Card Ethernet interface, and TCP/IP stack drivers which you can obtain from ARM (part number KPI-0014).

There are a number of links on an ARM Development Board that must be set up appropriately for use with Angel.

The memory REMAP link (LK18):

In: ROM at 0, can be remapped following startup.

Out: ROM at high address only, RAM at 0.

Ensure you set this link In for use with Angel running from ROM or Flash, and Out if EmbeddedICE is being used to download Angel into RAM before running it. This link determines whether ROM appears at 0 on reset.

The EPROM SELECT link field (LK6), Position 5–6 (second jumper nearest to the processor card, see ***Figure 6-2: EPROM SELECT link field (LK6) pinout*** on page 6-7):

In: Flash selected (device is read-write).

Out: EPROM selected (device is read-only).

This should be set correctly, according to whether you are using Flash or EPROM. This is particularly important if you intend to use the Flash download utility (see ***2.7.1 Flash downloading with ADW*** on page 2-31).

The EPROM SELECT link field (LK6), Position 7–8 (jumper nearest to the processor card):

In: 16 bit EPROM/Flash selected.

Open Access

Out: 8 bit EPROM/Flash selected.

This should be set correctly for the type of EPROM/Flash device. U13 holds a 16-bit device and U12 holds an 8-bit device.

If you have problems

If, for any reason, the debugger cannot communicate with the target board, quit the debugger and reset the board by pressing the red reset button (SW2).

2.5.2 Serial-only connection

Connect your host computer to the Serial A port on the ARM Development Board using a 9-pin RS232 cable. Power up the board (see ***2.3.3 Powering up the board*** on page 2-11).

Configuring the ARM Debugger for Windows

To configure the *ARM Debugger for Windows (ADW)* to access Angel rather than the ARMulator:

1. Start the ADW (not via the ARM Project Manager). In its default state, a message similar to the following is displayed:

```
ARMULATOR 2.0 [Jan 6 1997]ARM7DTMI, 15.0MHz, 4Gb, Dummy
  MMU, Soft Angel 1.4,[Angel SWIs, Demon SWIs],FPE,
  Profiler, Tracer, Little endian
```

2. Choose **Configure Debugger...** from the Options menu. The Debugger Configuration property sheet is displayed (see ***Figure 2-4: The Debugger Configuration property sheet*** on page 2-15).
3. Click on the **Target** tab.
4. Select **Remote_A**.
5. Click on **Configure...** and select **Serial**.
6. Select the serial communication port and line speed. (The ARM Development Board operates at 38400 bps.)
7. Click on **OK**.
8. If the target board being used runs in big-endian mode (the default is little-endian mode), click on the **Debugger** tab, and click on **Big** in the Endian selection.
9. Click on **OK**.

The debugger is restarted. The Restarting dialog box is displayed. Rapidly changing numbers are shown, indicating reading and writing to the target.

Note *If the target board does not respond (the numbers in the Restarting dialog box do not change), ADP times out within approximately five seconds and defaults back to ARMulator.*

When the configuration has completed successfully, a message similar to the following is displayed in your host console window:

```
EmbeddedICE Manager (ADP ARM7TDI) 2.01 (Advanced RISC Machines)
```


Open Access

Configuring armsd

To operate armsd with Angel using a serial connection only, type:

```
armsd -li -adp -port s=1 -linespeed 38400
```

where:

`-bi`	is the little endian switch. Use `-bi` for big-endian.
`-adp`	is the switch to allow communication using ARM Debug Protocol required by Angel.
`-p s=1`	indicates that the serial port is port 1 on the host computer. You can type the serial port number (1 or 2) or the host device name.
`-linespeed 38400`	sets the baud rate for serial communications between the host computer and the Development System.

When you enter this command, the host displays the debugger's opening banner, which will be similar to:

```
A.R.M. Source-level Debugger vsn 4.46 (Advanced RISC Machines
SDT2.10) [Dec 13 1996]
Angel Debug Monitor for PID (Built with Serial(x1), Parallel,
DCC) 1.00 (Advanced RISC Machines SDT 2.10)
Angel Debug Monitor rebuilt on Jan 20 1997 at 02:33:43
```

Note *You can view the options available for armsd by typing* `armsd -h` *at the command line.*

2.5.3 Serial/parallel connection

The ARM Development Board requires an adaptor card to connect to a standard parallel cable. Although a standard parallel cable connects to the development board as supplied, parallel communications to the host computer do not work. The adaptor card plugs into the development board's parallel port, and requires a ribbon cable connection between the adaptor card and on the development board. If you do not have an adaptor card and you need to use the parallel port, contact your development board supplier to obtain one, or contact ARM directly on the World Wide Web, at URL http://www.arm.com.

You will need the following:

- parallel port adapter (HBI-0021B)
- 20-way ribbon cable (HCI-0008)
- serial cable, 9F/25M D-Sub to 9F D-Sub (HCI-0004)
- parallel cable, 25M D-Sub to 25M D-Sub (HCI-0005)
- ARM Software Development Toolkit v2.10 or later
- Angel in Flash or EPROM

Open Access

If you are using a Sun workstation or clone that has a 26-way mini D-Sub parallel port socket, you also need the following cable:

- Sun parallel cable, 26M 0.050" slimline D-Sub to 25F D-Sub (HCI-0009)

  The Sun parallel cable is to be used in conjunction with the standard parallel cable.

Connecting up the hardware

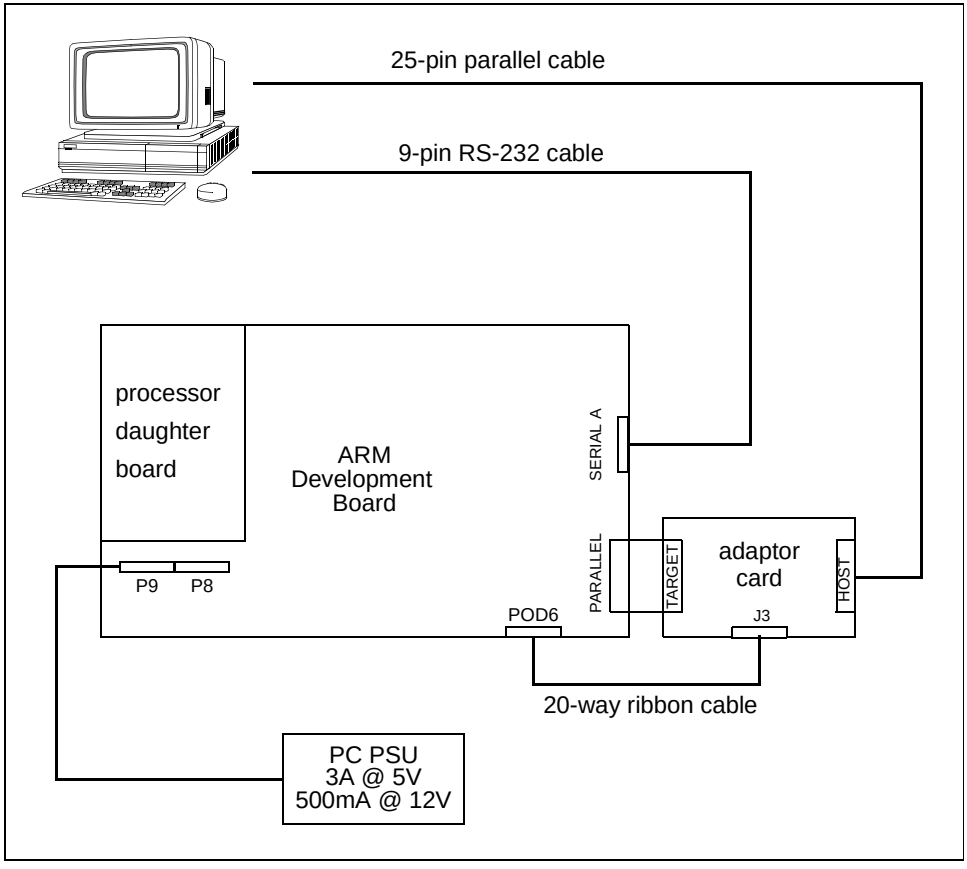


Figure 2-6: Connecting up with Angel using serial and parallel connections

1 Remove all the links of link field (LK11) and the INTERRUPT ENABLE link (LK10).

You cannot use the interrupt button (SW1), the LEDs (D7–D10), or the DIP switches (S3) in conjunction with the parallel port adaptor. This is because these features make use of the parallel port itself, which would clash with the parallel debugger communication. If you need to use the switches, LEDs, or interrupt button, connect to your host using either serial only or Ethernet connection.

Open Access

2 Plug the parallel port adaptor card TARGET plug (J1) into the parallel port socket (SK3) on the development board.

3 Connect the 20-way ribbon cable between connector (J3) on the adaptor and connector POD6 on the development board.

4 Connect the parallel cable between your host and the parallel port adaptor HOST plug (J2).

Some Sun workstations and clones have a mini 26-way parallel port connector. You need the Sun parallel cable (HCI-0009) in conjunction with the standard parallel cable to connect to this type of host.

5 Connect the serial cable between your host and serial port A (PL4) on the development board.

Configuring ADW

To configure the ADW to access Angel rather than the ARMulator:

1 Start the ADW (not via the ARM Project Manager). In its default state, a message similar to the following is displayed:

```
ARMULATOR 2.0 [Jan 6 1997]ARM7DTMI, 15.0MHz, 4Gb, Dummy
  MMU, Soft Angel 1.4,[Angel SWIs, Demon SWIs],FPE,
  Profiler, Tracer, Little endian
```

2 Choose **Configure Debugger...** from the Options menu. The Debugger Configuration property sheet is displayed (see ***Figure 2-4: The Debugger Configuration property sheet*** on page 2-15).

3 Click on the **Target** tab.

4 Select **Remote_A**.

5 Click on **Configure...** and select **Serial/Parallel**.

6 Select the serial and parallel communication ports, and the serial line speed. (The ARM Development Board operates at 38400 bps.)

7 Click on **OK**.

8 If the target board being used runs in big-endian mode, click on the **Debugger** tab, and click on **Big** in the Endian selection. (The default is little-endian mode.)

9 Click on **OK**.

The debugger is restarted. The Restarting dialog box is displayed. Rapidly changing numbers are shown, indicating reading and writing to the target.

Note *If the target board does not respond (the numbers on the Restarting dialog box do not change), ADP times out within approximately five seconds and defaults back to ARMulator.*

When the configuration has completed successfully, a message similar to the following is displayed in your host console window:

```
EmbeddedICE Manager (ADP ARM7TDI) 2.01 (Advanced RISC Machines)
```


Open Access

Configuring armsd

Note *The parallel port must be configured for standard communication during all download operations when communicating with the target development system in serial/parallel mode. See* ***6.6 Parallel Port/LED mode configuration*** on page 6-9 *for details.*

To operate armsd with Angel using serial and parallel connections, type:

```
armsd -li -adp -port s=1,p=1 -linespeed 38400
```

where:

`-li`	is the little-endian switch. Use `-bi` for big-endian.
`-adp`	is the switch to allow communication using the ARM Debug Protocol required by Angel.
`-port s=1,p=1`	indicates that the serial port is port 1 on the host computer, and the parallel port is port 1. The serial and parallel ports can be shown numerically (1 or 2) or be the host device name required.
`-linespeed 38400`	sets the baud rate for serial communications between the host computer and the Development System.

When you enter the above command, the host displays the debugger's opening banner, which will be similar to:

```
A.R.M. Source-level Debugger vsn 4.46 (Advanced RISC Machines
SDT2.10) [Dec 13 1996]
Angel Debug Monitor for PID (Built with Serial(x1), Parallel,
DCC) 1.00 (Advanced RISC Machines SDT 2.10)
Angel Debug Monitor rebuilt on Jan 20 1997 at 02:33:43
```

Note *You can view the options available for armsd typing* `armsd -h` *at the command line.*

Features of the parallel port adaptor

The development board (HBI-0011B) has a nonstandard parallel port pinout. You can use the parallel port adaptor to enable parallel download, and to convert the development board parallel port to the same pinout as a standard PC parallel port.

The parallel port adaptor has two modes of operation. These modes are selected by the link (LK1). When the link is in place, the adaptor is configured for download. When the link is removed, the adaptor is configured to appear as a PC parallel port with the correct pinout. Thus, a printer or similar peripheral could be driven from the female connector (J2).

The adaptor card has an LED marked BUSY. This is connected to the host busy line, and during download this LED flashes on and off.

2.5.4 Ethernet connection

The ARM Development Board requires an Ethernet Adaptor Kit (KPI-0014) to connect to an Ethernet network. This contains:

- an Olicom Ethernet PCMCIA adaptor

Open Access

- Ethernet version of Angel ($3^1/_2$" DOS format disk) containing Pacific Softworks "Fusion" TCP/IP stack

The Fusion stack has been compiled to support five sockets, and the maximum size of IA and UDP packets is 2048 bytes. These limits result in a 22KB Fusion heap. The limits cannot be changed by users of Angel unless a source licence for Fusion is obtained.

If you are using Windows, use the install utility on the disk to install the software in your ARM210 directory. If you are using UNIX, follow the instructions in the readme file for information about where to copy the files.

With the ARM Development Board connected to the host via serial only (see ***2.5.2 Serial-only connection*** on page 2-21) or serial/parallel (see ***2.5.3 Serial/parallel connection*** on page 2-22) using Angel, or via EmbeddedICE (see ***2.4 Communicating with EmbeddedICE*** on page 2-13), use the Flash downloader facility to load the file `angel-E.rom`. The file is usually located in `ARM210\Angel\Images\pid\little`. See ***2.7 Using the Flash Downloader*** on page 2-31 for information about how to use the Flash downloader.

Configuring your IP address and net mask

With the same target–host connection as described above, use the Flash downloader program to override the default IP address and net mask used by Angel for Ethernet communication. Consult your system administrator for an IP address and net mask, and for help with connecting to the network.

If you are using ADW:

1. Select the **Load Image...** option from the File menu.
2. Check the **Download to Flash Memory** box.
3. Ensure that the file text box is blank.
4. Type `-e` in the **Arguments** text box.
5. Click on **OK**.

   You are prompted for the IP address and net mask in the console window.

If you are using armsd, type:

```
armsd -adp flash.li -e
```

The program prompts for the IP address and net mask.

Exit the debugger, and power down the board. Attach the Olicom PCMCIA Ethernet adaptor to the Olicom-supplied cable for connection to your Ethernet network. Plug the adaptor into either PC Card slot on the ARM Development Board. Ensure that the REMAP link (LK18) is connected.

Open Access

Configuring ADW

To configure the ADW to access a remote target rather than the ARMulator:

1. Start the ADW (not via the ARM Project Manager). In its default state, a message similar to the following is displayed:

```
ARMULATOR 2.0 [Jan 6 1997]ARM7DTMI, 15.0MHz, 4Gb, Dummy
  MMU, Soft Angel 1.4,[Angel SWIs, Demon SWIs],FPE,
  Profiler, Tracer, Little endian
```

2. Choose **Configure Debugger...** from the Options menu. The Debugger Configuration property sheet is displayed.
3. Click on the **Target** tab.
4. Select **Remote_A**.
5. Click on **Configure...** and select **Ethernet**.
6. Enter your IP Address (as configured on your target).
7. Click on **OK**.
8. If the target board runs in big-endian mode (the default is little-endian mode), click on the **Debugger** tab, and click on **Big** in the Endian selection.
9. Click on **OK**.

The debugger is restarted. The Restarting dialog box is displayed. Rapidly changing numbers are shown, indicating reading and writing to the target.

Note *If the target board does not respond (the numbers in the Restarting dialog box do not change), ADP times out within approximately five seconds and defaults back to ARMulator.*

When the configuration has completed successfully, a message similar to the following is displayed in your host console window:

```
Angel Debug Monitor for PID (Built with Serial(x1), Parallel,
Ethernet, DCC) 1.00 (Advanced RISC Machines SDT 2.10)
Angel Debug Monitor rebuilt on Jan 20 1997 at 02:33:43
```

Configuring armsd

To operate armsd with Angel using Ethernet connection, type:

```
armsd -li -adp -port e=address
```

where:

`-li`	is the little-endian switch. Use `-bi` for big-endian.
`-adp`	is the switch to allow communication using the ARM Debug Protocol required by Angel.
`-e`	is the Ethernet communications switch.
`address`	is the ethernet address configured in the target system.

Open Access

When you enter the above command, the host displays the debugger's opening banner, which will be similar to:

```
A.R.M. Source-level Debugger vsn 4.46 (Advanced RISC Machines
SDT2.10) [Dec 13 1996]
Angel Debug Monitor for PID (Built with Serial(x1), Parallel,
Ethernet, DCC) 1.00 (Advanced RISC Machines SDT 2.10)
Angel Debug Monitor rebuilt on Jan 20 1997 at 02:33:43
```

Note *You can view the options available for armsd typing* `armsd -h` *at the command line.*

Open Access

2.6 Communicating with Angel using EmbeddedICE

The Debug Comms Channel provided by EmbeddedICE can be used by Angel for communication with the host debugger. This is known as ADP over JTAG.

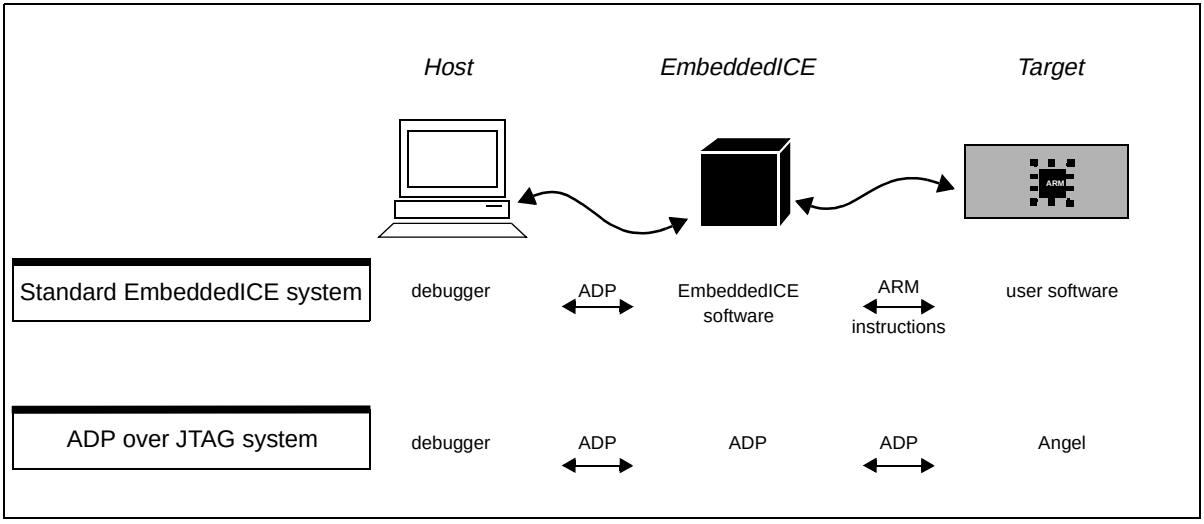


Figure 2-7: Debug Comms Channel, Angel, and EmbeddedICE

In a standard EmbeddedICE system, the EmbeddedICE software acts as the Debug Agent that processes debug requests issued by the host. EmbeddedICE interacts with the target processor by halting execution (entering debug state) and then forcing the ARM processor to run instructions that read and write the information the debugger requires. As a result, the ARM processor is halted for significant periods of time, which can cause problems for real-time applications.

An alternative way of using EmbeddedICE is as a communication link, provided by the Debug Comms Channel. This enables the debugging of real-time systems that have no other communication ports. Angel is required on the target, and uses the Debug Comms Channel in the same way as it would a serial link (see ***Figure 2-7: Debug Comms Channel, Angel, and EmbeddedICE***). The processor is never put into debug state, so this system may be more suitable for real-time applications.

To use this system, a version of Angel that supports the Debug Comms Channel is required on the target system (the default), and a different ROM image for EmbeddedICE interface is also necessary (this is the ADP over JTAG software).

The ADP over JTAG software is in `ARM210\Angel\Images\iceagent\adpjtag.rom`. To download it into the EmbeddedICE interface after power on or reset:

- if you are using armsd, use the `loadagent` command

Open Access

- if you are using ADW, use the **Load Agent...** option on the EmbeddedICE Configuration dialog box (see ***Figure 2-5: The EmbeddedICE Configuration dialog box*** on page 2-17)

Alternatively, you can program a new EPROM and plug it into EmbeddedICE. You can obtain the latest version of ADP over JTAG from http://www.arm.com.

The ARM debuggers can be used in exactly the same way as when they are connected to a target running Angel over a serial link.

For more information about downloading a new debug agent, see the *ARM Software Development Toolkit User Guide* (ARM DUI 0040).

Open Access

2.7 Using the Flash Downloader

See ***7.8 Flash Downloader*** on page 7-24 for further information about the Flash downloader and the supported devices.

Ensure you have your board configured correctly for your Flash device. See ***6.5 Setting the EPROM/Flash Speed and Width*** on page 6-7.

2.7.1 Flash downloading with ADW

Downloading into Flash overwrites your existing Angel image. Ensure that the download completes successfully before you reset or power off the board.

The Flash downloader is a utility provided with the ARM Software Development Toolkit version 2.10 or later, and integrated into the ARM Debugger for Windows (see ***7.8 Flash Downloader*** on page 7-24 for Flash device basics and a worked example). You can use it to program a Flash memory device on the board. This works only if Angel is running from RAM (by default Angel copies itself to RAM and runs from RAM), or if EmbeddedICE is being used rather than Angel. The correct version for the endian configuration of the board should be used. (See ***2.7.4 Big- and little-endian operation***.)

To download a new image:

1. Select **Load Image...** from the File menu.
2. Check the **Download to Flash Memory** box.
3. Click on **Browse**, select the required ROM image (for example `ARM210\Angel\Images\pid\little\angel.rom`) and click on **OK**.

   The Flash downloader attempts to recognize the Flash device being used, and fails if it does not recognize it, because it must understand how to program it.

   The supported Flash devices are shown in ***Table 2-2: Supported Flash devices***.

Manufacturer	Part number	Width	Socket
ATMEL	AT29C040A	8-bit	U12
ATMEL	AT29C1024	16-bit	U13

Table 2-2: Supported Flash devices

4. You are prompted in the console window for the sector from where the programming should start. Angel should be programmed into the start of the Flash, (from sector 0). Click on the console window, and press Return.

   On completion, the following message is displayed in the console window:

   ```
   Flash written and verified successfully
   ```


Open Access

2.7.2 Flash downloading with armsd

To download a new version of Angel to the Flash device using armsd, type:

```
armsd [options] flash[endianness] filename
```

where:

`[options]` depend on the debugging system you are using (see ***2.4.3 Configuring the armsd debugger to use EmbeddedICE*** on page 2-18 or the relevant subsections in ***2.5 Communicating with Angel***)

`[endianness]` specifies the endianness of the image:

- `.li` for little-endian
- `.bi` for big-endian

`filename` for example:

```
ARM210\Angel\Image\pid\little\angel.rom
```

To download the code to the Flash device, type `go` at the `armsd:` prompt.

On completion, the following message is displayed:

```
Flash written and verified successfully
```

2.7.3 Remapping to run Angel from Flash

To use the code as boot code, the system has to be set up for power-on remap using the REMAP link LK18, which is at the top of the main board, above and to the right of the frequency select switch S1. When this link is connected, the memory map changes at power-on to move the lowest part of ROM into the base of memory. This means that any code in ROM runs immediately at power-on.

Make the connection at LK18 and remove the EmbeddedICE JTAG connection,. This leaves the board in stand-alone mode.

When the board is next powered up, the code runs as described above. To restart the code, press the RESET button (the red-capped button between parallel and serial port connectors).

2.7.4 Big- and little-endian operation

If you are running a little-endian version of Angel and wish to change to a big-endian version, you can do this by running the Flash download program. However, because of an incompatibility between the way big- and little-endian code is stored in 16-bit wide devices, this works only with an 8-bit Flash device.

1. Ensure you are using the 8-bit Flash device (U12).
2. Start little-endian Angel by switching on the board and connecting to the debugger.
3. Run the Flash download program and program the Flash with the big-endian Angel image. This works because Angel operates out of SRAM.

Open Access

4 Quit the debugger and switch off the board.

5 Change the EPROM controller (U10) to be the big-endian controller.

6 Insert the BIGEND link (LK4).

7 Power up the board and connect the debugger. Ensure the debugger is configured for big-endian operation.

When you have a big-endian Angel in Flash, you can use a big-endian version of the Flash downloader to program a new copy of Angel into the 16-bit device. To do this:

1 Switch on the board.

2 Start the debugger.

3 Insert the link in the SELECT EPROM link field (LK6) Position 7-8 to select a 16-bit Flash device.

Remember that you must provide a 16-bit wide Flash device, because one is not supplied with the board.

Note *There is no performance gain if you use a 16-bit wide device, because Angel copies itself to SRAM and executes from there.*

Open Access

Open Access

3 A Worked Example

3.1	Getting Started with the Example Code	3-2
3.2	Using the Code with the Windows Toolkit	3-3
3.3	Using the Code on the Command Line	3-7

Open Access

A Worked Example

3.1 Getting Started with the Example Code

This chapter describes how you use some example code on the ARM Development Board. An example for compiling and running code to drive the board's parallel port (leds.c) is provided for both Windows toolkit and command-line operation.

For full descriptions of how to use the ARM Software Development Toolkit (SDT) see the *ARM Software Development Toolkit User Guide* (ARM DUI 0040) and the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041).

For the "leds" example to work, all links in the link field (LK11), in the bottom right-hand corner of the board must be inserted. The parallel communication port cannot be used in the configuration. Set up your target system communicating with EmbeddedICE or Angel (serial-only or Ethernet connection), and exit the debugger. See ***Chapter 2, Setting up your Target System***.

Note 1 *If your system crashes during debugging, it is advisable to quit from the debugger, reset the board, and then the EmbeddedICE (if you are using it), by pressing the red button on each unit. Then restart the debugger.*

Note 2 *Code compiled using the Software Development Toolkit prior to version 2.10 does not run if you are using Angel. See Application Note 43:* Design Considerations when using Angel Debug Monitors *(ARM DAI 0043) for further information.*

Open Access

3.2 Using the Code with the Windows Toolkit

The following notes are intended as a guide for testing the target system setup.

3.2.1 Using the ARM Project Manager

ARM Project Files (.apj) are supplied with each module of example code. These Project Files set up the Project Manager to compile, link and run the example code.

To open a project file for the “leds” example:

1. Start the ARM Project Manager.
2. Select **Open** from the File menu, then use the browser to find the project directory. (This is usually `ARM210\Examples\apm\leds`.)
3. Click on the file called `leds.apj`. This is the project file supplied for the “leds” example.

   `leds.apj` is shown in the **Filename** dialog box.
4. Click the **Open** button.

You have now opened the project for the “leds” example. ***Figure 3-1: The leds project window*** on page 3-4 shows how the project looks when you have expanded all branches by clicking on the plus (+) signs.

To build a debug variant and create a target image suitable for debug:

1. If the **Build Debug** button is not visible at the bottom of the project window, select the **Debug** variant in the project window.
2. Click **Build Debug** button, or select **Build leds.apj "Debug"** from the Project menu.

The build status is now reflected in the Build log, displayed at the bottom of the project window. You should not receive any warnings or errors. The last line in the log should read:

```
i Information: "c:\ARM210\Examples\apm\leds\leds.apj";
   Project is up to date
```

In the project window, leds.c, leds.o, and leds.axf should be marked with blue ticks.

Open Access

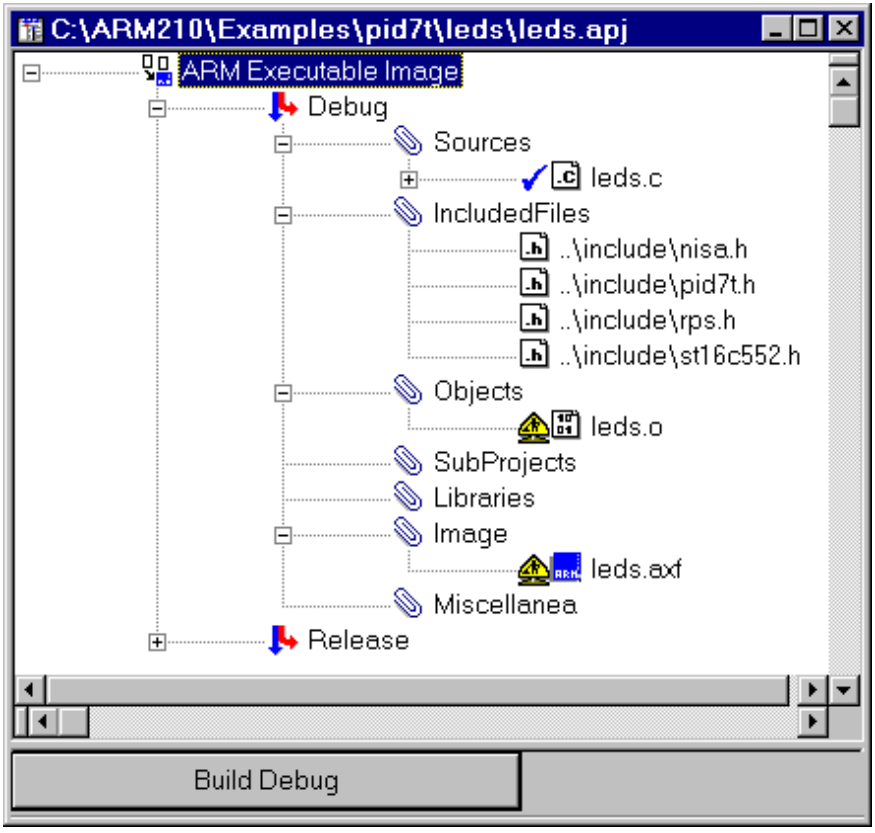


Figure 3-1: The leds project window

3.2.2 Using the ARM Debugger for Windows

To launch the debugger and execute the “leds” example:

1. Click the **Debug** button on the toolbar, or select **Debug leds.apj** from the Project menu.
2. If you have the debugger configured correctly, you are asked whether you want to start in remote debugging. Click on **Yes**.

   If you are not prompted, see ***Chapter 2, Setting up your Target System***.

   The project window should be similar to ***Figure 3-2: Debugging leds.apj***.

Open Access

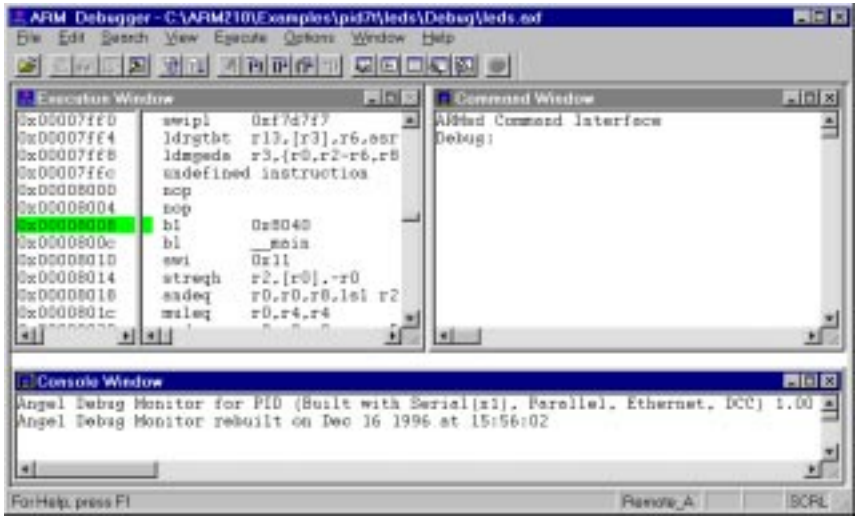

ARM Debugger - C:\ARM210\Examples\pid7t\leds\Debug\leds.axf
File Edit Search View Execute Options Window Help
Execution Window
Command Window
ARMul Command Interface
Debug:
Console Window
Angel Debug Monitor for PID (Built with Serial[s1], Parallel, Ethernet, DCC) 1.00
Angel Debug Monitor rebuilt on Dec 16 1996 at 15:56:02
For Help, press F1
Remote_A
SCRL

Figure 3-2: Debugging leds.apj

3 Click the **Go** button on the toolbar, or select **Go** from the Execute menu. The debugger automatically breakpoints on the first instruction following `main`. The Execute window should look similar to ***Figure 3-3: First breakpoint***.

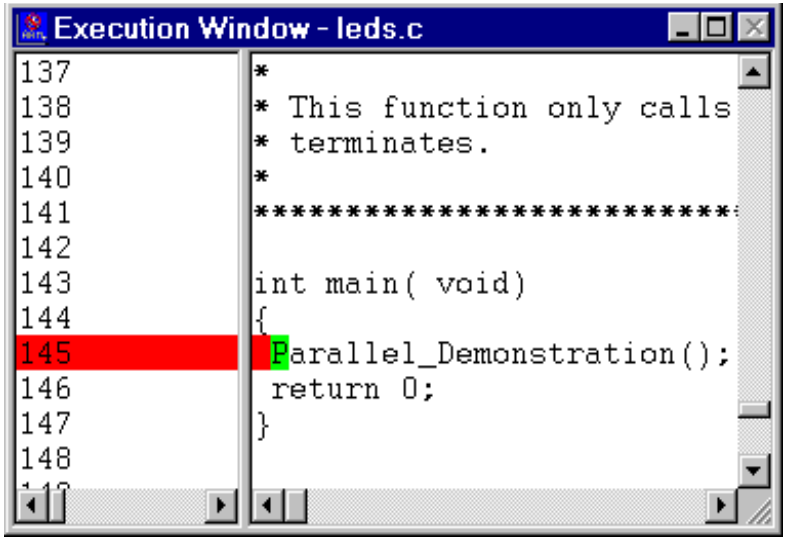


Figure 3-3: First breakpoint

Open Access

4 Click the **Go** button again.

The code executes and the four LEDs (pp0–4) flash sequentially. You can change the speed of the LED sequence by changing the settings of the DIP switch S3, which is positioned in the bottom right-hand corner of the development board, just below the 16C552 device (U21).

The code runs in an infinite loop, and so continually executes until you interrupt it by clicking the **Stop** button on the toolbar.

The Debugger offers many other debugging features. See the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for more information.

Open Access

3.3 Using the Code on the Command Line

ARM Development Boards are primarily designed to develop Thumb code and to develop system components around the ARM core. However, they can operate with either Thumb or ARM code or even a mixture of both.

3.3.1 Compiling the code

To compile leds.c into Thumb code, use the Thumb-specific compiler. From the directory `ARM210\Examples\apm\leds`, type the command:

```
tcc leds.c -o leds -i ..\include
```

The `-o leds` option generates an image file with the name “leds”. The `-i` option is a pointer to the required include file directory.

To compile `leds.c` into ARM code, use the command:

```
armcc leds.c -o leds -i ..\include
```

If you do not use the `-o` option with the `tcc` or `armcc` command, the final image file automatically takes the name of the single file (leds in this example).

The working directory now has two new files, the object file (`leds.o`) and the image file (`leds`) which will be downloaded and run on the target system.

3.3.2 Running the code on the development board

You use the ARM debugger, armsd, to download and run the generated code image. This allows the code to be emulated in the local host using a software model of the ARM processor, or to be downloaded and run on the development board.

See ***Chapter 2, Setting up your Target System*** for information about the operation of the development board with each of the different debug systems and connections.

To start the debug monitor from the command line, for debugging on a remote target:

1 Type the command:

```
armsd -li -port s=1,p=1 -linespeed 38400]
```

where:

`-li`	is the little-endian switch. Use `-bi` for big-endian.
`-port s=1,p=1`	indicates that the serial port is port 1 on the host computer, and the parallel port is port 1. Use: `-port e-`*`address`* for Ethernet connection.
`-linespeed`	changes the serial baud rate of the link. The default is 9600 baud. The recommended rate is 38400.

Open Access

This command initializes the debug monitor, which then displays:

```
A.R.M. Source-level Debugger vsn 4.46 (Advanced RISC Machines
       SDT 2.10) [Dec 13 1996]
EmbeddedICE Manager (ADP ARM7TDI) 2.01 (Advanced RISC
       Machines)
armsd:
```

where `armsd:` is the debugger command-line prompt.

2 Load the code image by typing the following at the prompt:

```
load leds
```

When the code image has been loaded, the system displays the command-line prompt.

3 Execute the code by typing the following at the prompt:

```
go
```

The code executes and the four LEDs (pp0–4) flash sequentially. You can change the speed of the LED sequence by changing the settings of switch S3, which is positioned in the bottom right-hand corner of the development board, just below the 16C552 device.

The code runs in an infinite loop, and so continually executes until you interrupt it. To send the break command to the EmbeddedICE unit, hold down the Control key and press the ‘C’ key. The monitor displays an interrupt message, for example:

```
^CInterrupted at PC = 0x00008120 (Parallel_Demonstration + 0x54)
+0054 0x00008120: 0xdafc .. : bge 0x0000811c
armsd:
```

where:

`^C`	This is an echo of the break command sent.
`interrupted at PC = ...`	This shows the address at which the code was interrupted.
`(Parallel_Demo...)`	This shows the routine executed and offset of the current line. The following line is a disassembly of the next assembler instruction due to be executed.

The debugger offers many other debugging features. See the *ARM Software Development Toolkit Reference Guide* (ARM DUI 0041) for more information.

To exit from the debugger, type at the prompt:

```
quit
```

Control is returned to the host system command line.

Open Access

4 Test Equipment

4.1	Introduction to Real-Time Trace	4-2
4.2	The Test Interface	4-3

Open Access

Test Equipment

4.1 Introduction to Real-Time Trace

Usually, you can download and debug code using the ARM toolkit in combination with either a debug monitor resident on the board, or an EmbeddedICE interface. However, it is sometimes necessary to use a logic analyzer so that code execution can be traced in real-time.

ARM HP Inverse Assembler

ARM have developed an Inverse Assembler for use with HP logic analyzers to enable disassembly of the code captured in the trace buffer. If you are using an ARM Development Board with the *Advanced Microcontroller Bus Architecture (AMBA) Advanced System Bus (ASB)* signals available, you can achieve full code disassembly by using the ARM HP Inverse Assembler. A set of six 20-way box headers are provided on the ARM Development Board for this purpose (POD 7–12). A further set are mounted on the ARMTDMI daughter board. See the ARM *HP Inverse Assembler User Guide* (ARM DUI 0062A) for more information on the inverse assembler.

In-Circuit Emulation

Alternatively, you can replace the AMBA master (processor) with an *in-circuit emulator (ICE)*. A variety of ICE strategies are implemented on the ARM Development Board, which is why the processor is mounted on a daughter board. By removing the daughter board and replacing it with a module that emulates its functionality, it is possible to perform real-time in-circuit emulation.

ARM is working with a number of third parties to develop ICEs that can be used with ARM Development Boards. For further information, please contact:

Lauterbach Datentechnik GmbH
Fichtenstr. 27
D85649 Hofolding
Tel: ++49-8104-8943-0
Fax: ++49-8104-8943-49
E-mail: info@lauterbach.com
WWW: http://www.lauterbach.com

Lauterbach Inc.
945 Concord Street
Framingham, MA 01701
Tel: (508) 620 4521
Fax: (508) 620 4522
E-mail: info@lauterbach.com

Yogakawa Digital Computer Corporation
Musashino YS Bldg. 1-19-3 Nakacho
Musashino-shi, Tokyo, 180 Japan
Tel: +81-422-56-9101
Fax: +81-422-56-9102

Orion Instruments Inc
1376 Borregas Avenue
Sunnyvale, CA 94089-1004
Tel: (408) 747-0440
Fax: (408) 747-0688
E-mail: info@oritools.com
WWW: http://www.oritools.com

Open Access

4.2 The Test Interface

The test interface is the default ASB master. If no other master requests the bus, the arbiter grants access to the *test interface controller (TIC)*. The test interface comprises a controller and a number of bidirectional latched transceivers that control flow between the test bus and ASB.

Insert the USETIC link (LK17) to enable the test interface.

Refer to the *AMBA Specification (*ARM IHI 0001*)* for details of the test port provided.

The following external signals are defined:

T_CLK	test clock	input
T_REQA	test bus request A	input
T_REQB	test bus request B	input
T_ACK	test acknowledge	output
T_D[31:0]	test bus	bidirectional

Using the test bus interface, it is possible to gain control of the ASB in real-time, and perform manufacturing test and in-circuit diagnostics. On a circuit board this is not usually a problem. However, an ARM Development Board is a board-level implementation of a typical ASIC in which it is not possible to examine the internal connections.

Alternatively, you can use the test interface to test multiple master AMBA systems.

4.2.1 Connecting external equipment to the test bus

You need additional equipment to drive the test interface on the ARM Development Board, typically a parallel I/O device offering 36 separate input/output lines. You connect the equipment to the board via two 20-way box headers mounted on the right-hand edge of the development board, called TEST1 and TEST2

Refer to the relevant hardware guide for your development board for details of how the connector pins are assigned.

Open Access

Test Equipment

Open Access

5 Programmer's Model of the ARM Development Board

5.1	Introduction	5-2
5.2	Development Board Architecture	5-3
5.3	Main Memory Map	5-7
5.4	RAM Region	5-9
5.5	ROM Region	5-12
5.6	Reference Peripherals	5-13
5.7	Serial, Parallel, and PC Card Ports	5-24

Open Access

Programmer's Model of the ARM Development Board

5.1 Introduction

This chapter provides an overview of the ARM Development Board architecture, a guide to the memory and I/O map, and information about how to program the board.

The following are useful reference documents. You should refer to these to understand the functionality of AMBA modules.

- *AMBA Specification* (ARM IHI 0001)
- *Reference Peripherals Specification* (ARM DDI 0062D)

Open Access

5.2 Development Board Architecture

The ARM Development Board is an implementation of an example AMBA system. It comprises an ARM processor connected via the *Advanced System Bus* (*ASB*) to fast synchronous SRAM, an external bus interface, and a bridge to the slower, low-power *Advanced Peripheral Bus* (*APB*), see ***Figure 5-1: Schematic of the ARM Development Board***.

You can think of the ARM Development Board as a microcontroller system with its support peripherals constructed from discrete devices. The bus master, system modules, APB bridge and peripherals, SRAM, and external bus interfaces form the heart of a microcontroller, and are usually integrated onto a single chip. Additional resources such as memory, and peripherals like the PC Card (PCMCIA) and serial and parallel ports may also be integrated or interfaced to externally.

Figure 5-1: Schematic of the ARM Development Board shows the following functional blocks:

- AMBA bus master comprising an ARM processor and *programmable logic device (PLD)*
- AMBA system modules, arbiter and decoder
- "on-chip" (synchronous SRAM) memory
- SRAM
- EPROM or Flash
- DRAM
- test interface
- APB bridge
- APB peripherals (timer, interrupt controller, and remap/pause controller)
- ASB expansion connectors
- APB expansion connectors
- NISA bus bridge
- PC Card (PCMCIA)
- serial and parallel ports

Open Access

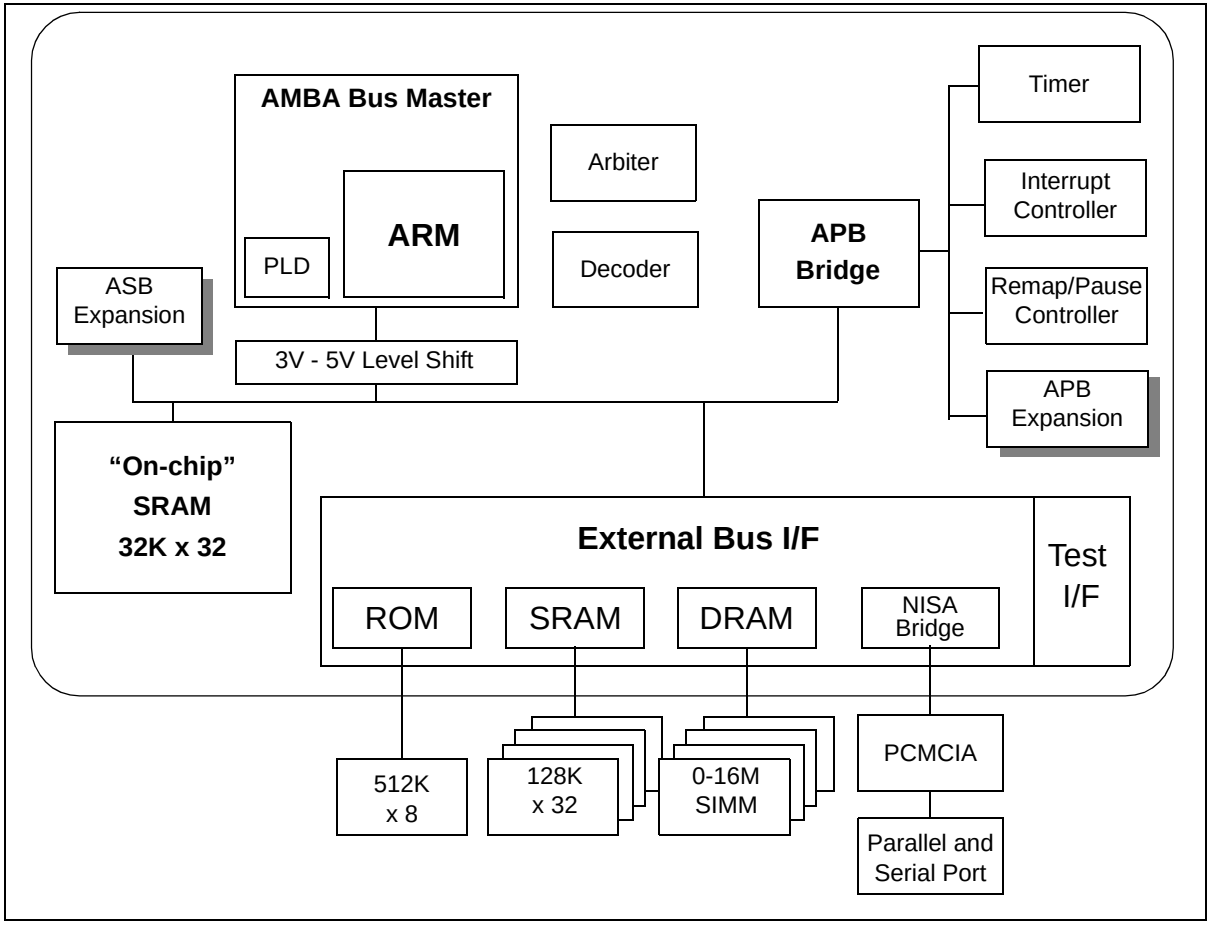


Figure 5-1: Schematic of the ARM Development Board

Open Access

5.2.1 Memory overview

The ARM Development Board can be configured with the types of memory shown in ***Table 5-1: Memory types and sizes***.

Memory Type	Size
SSRAM	128KB
SRAM	512KB
DRAM	Up to 16MB
EPROM/Flash	Up to 512KB

Table 5-1: Memory types and sizes

Bank size and cycle times can be configured. The memory types are described later in ***5.4 RAM Region*** on page 5-9 and ***5.5 ROM Region*** on page 5-12.

PC Card

You can add additional memory using the PC Card (PCMCIA) slots. Two slots are provided using the Vadem VG-468 PCMCIA Interface Controller. You can use any of the wide range of memory and disk cards available. To use PC Cards with the ARM Development Board, you must write your own software (see ***7.6 PC Card (PCMCIA) SRAM Memory Card Read/ Write Example*** on page 7-14).

5.2.2 I/O peripherals overview

The board architecture supports peripherals in two areas:

- on the *Advanced Peripheral Bus (APB)*
- on the NISA (not ISA) bus

An interrupt controller, counter/timer, and remap/pause controller implement a system-specific version of the Reference Peripheral Specification described in ***5.6 Reference Peripherals*** on page 5-13. These are placed on the APB.

A double PC Card slot interface, two serial ports, and one parallel port are placed on the NISA bus, described in ***5.7 Serial, Parallel, and PC Card Ports*** on page 5-24.

Expansion connectors are provided for both the ASB and APB for the addition of extra peripherals. Refer to the hardware guide supplied with your development board for details.

Open Access

5.2.3 Big- and little-endian operation

The board is designed to work in big- or little-endian byte ordering, with minor configuration changes (see ***6.7 Big-endian Operation*** on page 6-12). The default is little-endian.

Register addresses are all word-aligned, with the exception of the PC Card interface. See ***7.6.2 Accessing the Vadem PCMCIA controller*** on page 7-16.

Open Access

5.3 Main Memory Map

The memory map has two states:

- Reset, following the application of a cold reset
- Normal, after the Address Remap register has been written to

In the Normal (remapped) configuration, aliases to regions of the RAM are provided from address 0x0 to 0x00FFFFFF. This provides a contiguous region of memory using a selection of RAM technologies. In the Reset configuration, the ROM is mapped into this alias space with the RAM devices accessible at higher addresses.

The top level breakdown of these two states is shown in ***Figure 5-2: Memory map***.

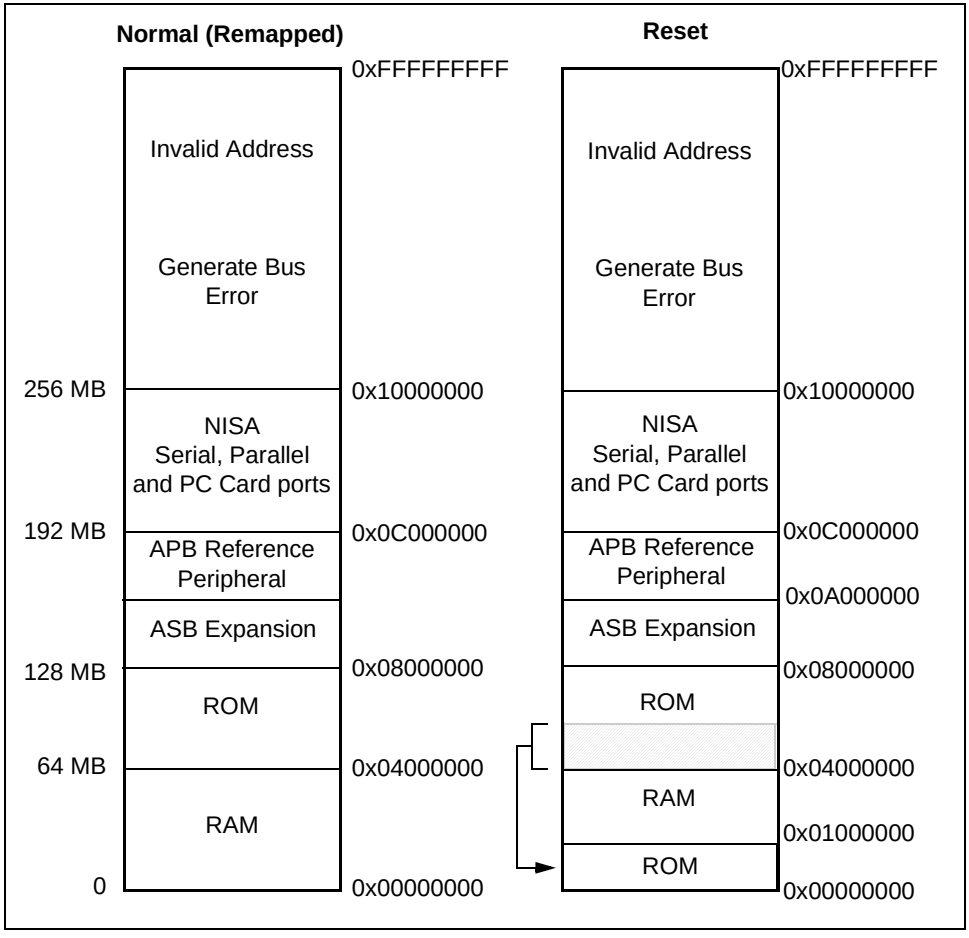


Figure 5-2: Memory map

Open Access

Note *When accesses to memory space 0x10000000 and above occur, the processor takes the ABORT exception.*

Open Access

5.4 RAM Region

The RAM region is split into four main blocks. Separate 16MB sections are reserved for DRAM, SRAM, and SSRAM, as shown in ***Figure 5-3: RAM region***.

The fourth 16MB block (starting at 0x0) is either an alias pointing to the base of the ROM region (in the Reset configuration) or an overlaid region of RAM (in the Normal (Remapped) configuration).

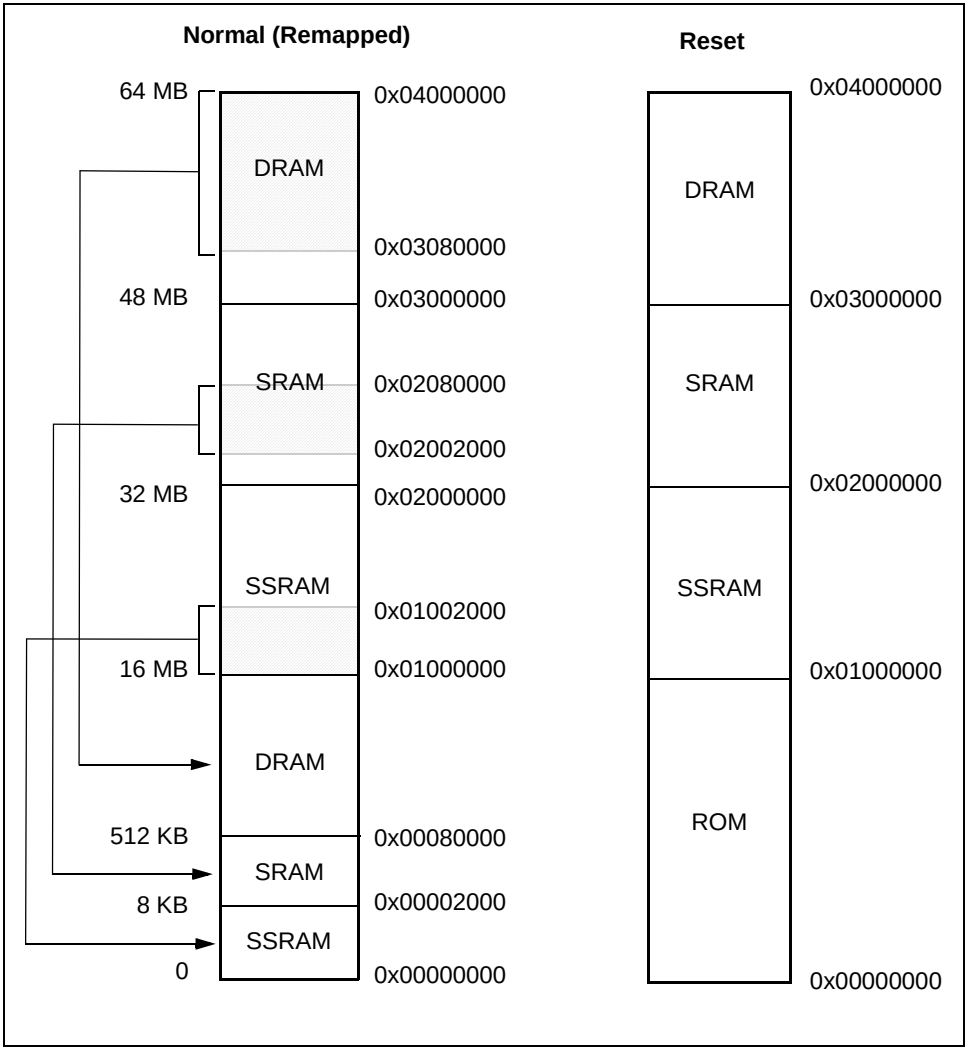


Figure 5-3: RAM region

Open Access

Normal (Remapped)		Reset	
Address	**Alias for**	**Address**	**Alias for**
0x00000000	0x01000000	0x00000000	0x04000000
0x00002000	0x02002000		
0x00080000	0x03080000		

Table 5-2: Aliases for the RAM region

With the REMAP link (LK18) in, at power up the board is in the Reset configuration, by default. In this state, the EPROM or Flash are at the bottom of the address map. If the remap register ClearResetMap is written to (see ***5.6.3 Remap and pause*** on page 5-22), the **REMAP** signal goes HIGH and the decoder switches to the Normal memory map in which there is RAM at the bottom.

If you are powering up a system in which there is no EPROM or Flash, or if you want to boot from RAM, you must remove the REMAP link. The **REMAP** signal is then always HIGH, and the board starts up with SSRAM at the bottom of the address map.

SSRAM

Synchronous SRAM (SSRAM) is used to provide single cycle memory. The SSRAM device is organized as 32KB x 32-bits and is chosen to represent "on-chip" memory. Use this area for time-critical routines, such as interrupt handlers. The SSRAM can usually be found at address zero. No configuration of this memory is required.

SRAM

Fast SRAM is used to emulate various memory schemes. The SRAM controller allows the four physical 128KB x 8-bit devices to be split into two logical banks of 256KB each. Each of these logical banks can be configured as 8-, 16- or 32-bit wide memory. The SRAM controller simulates these memory systems by inserting the correct number of wait states. Slow memory can also be simulated by specifying the number of bus cycles (2, 3, 4, or 5) required for a memory access. See ***6.4 Setting the SRAM Speed and Width*** on page 6-6.

DRAM

The DRAM controller supports one or two 72-pin SIMM slots and uses automatic SIMM presence-detection for contiguous memory configuration. The DRAM controller provides:

- fast page-mode burst mode sequential access support
- byte, halfword and word transaction support
- 256-byte boundary page-mode cycle break-up
- DRAM refresh controller (using CAS-before-RAS Refresh mode)

Open Access

- automatic module-size reconfiguration
- support for 4MB, 8MB,16MB, and 32MB SIMM modules

Note *Only 16MB of address space is allocated to DRAM on the standard board. Excess memory is ignored.*

The DRAM controller supports 72-pin modules built using 4Mb and 16Mb technology DRAM devices, with a RAS Access Time specified at 70ns or faster, and symmetric addresses (equal numbers of row and column addresses).

The following size DRAM SIMM modules are supported:

- 4MB 32x1M, 72-pin, 70ns (single-sided module, 8 x 1Mx4)
- 4MB 36x1M, 72-pin, 70ns (single-sided module, 9 x 1Mx4)
- 8MB 32x2M, 72-pin, 70ns (double-sided module, 2 x 8 x 1Mx4), 70ns
- 8MB 36x2M, 72-pin, 70ns (double-sided module, 2 x 9 x 1Mx4), 70ns
- 16MB 32x4M, 72-pin, 70ns (single-sided module, 8 x 4Mx4), 70ns
- 16MB 36x4M, 72-pin, 70ns (single-sided module, 9 x 4Mx4), 70ns
- 32MB 32x8M, 72-pin, 70ns (double-sided module, 2 x 8 x 4Mx4)
- 32MB 32x8M, 72-pin, 70ns (double-sided module, 2 x 9x 4Mx4)

These SIMMs are standard commodity parts for desktop computers and workstations. Byte parity is not required or used.

Note *No configuration is required to use the DRAM, but Slot A must be populated first. If both slots are used, they must be fitted with the same size and configuration SIMMs.*

Open Access

5.5 ROM Region

A region is reserved for the ROM, so that it is always visible.

The ROM region is the same for both Normal and Reset configurations of the memory map, as shown in ***Figure 5-4: ROM region***.

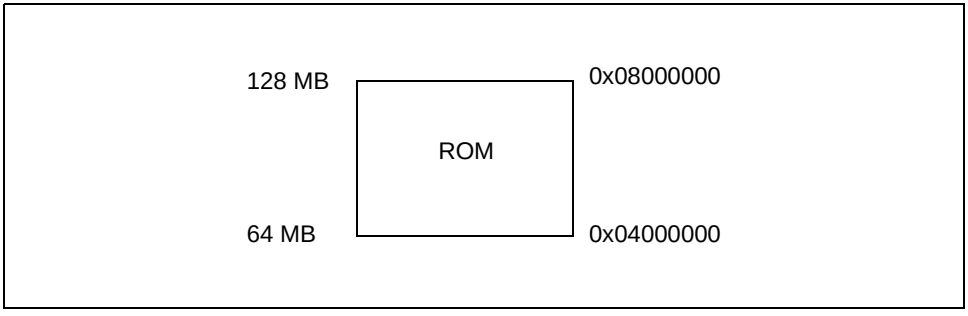


Figure 5-4: ROM region

Note *With the REMAP link (LK18) in, at power up the board is in the Reset configuration, and the ROM is visible from 0x0. A typical application located in this space then sets the remap register to alias RAM at the bottom of the address map.*

EPROM/Flash

Two sockets are provided, one for 8-bit wide memory and the other for 16-bit wide memory. The sockets can accept either standard EPROM or 5V-only Flash. The memory type, size and speed is configured by four links. See ***6.5 Setting the EPROM/Flash Speed and Width*** on page 6-7.

Open Access

5.6 Reference Peripherals

The reference peripherals are:

- interrupt controller (15 IRQ sources and one FIQ source)
- two 16-bit counter/timers with 8-bit prescalers
- remap and pause (standby) controller

This is a set of reference peripherals common to AMBA systems. For a full description of these, see the *ARM Reference Peripherals Specification* (ARM DDI 0062D).

The base addresses shown in ***Table 5-3: RPS base addresses*** have been implemented on the board:

Address	Name
0x0A000000	Interrupt Controller base
0x0A800000	Counter Timer base
0x0B000000	Remap and Pause Controller base
0x0B800000	Expansion (spare for user functions)

Table 5-3: RPS base addresses

A full address map of each peripheral is provided in the following sections.

All the reference peripherals are implemented in a single Xilinx *field programmable gate array (FPGA)*. When the FPGA is successfully configured, the green LED marked "FPGA OK" lights. If it does not, check that the 8-pin serial PROM (U28) and the adjacent links (LK16) are:

Name	Default
INIT	out
MODE 0	in
MODE 1	in
MODE 2	in

Table 5-4: Serial PROM and adjacent links

Open Access

Note *The FPGA can be reprogrammed by replacing the serial PROM or using a special download cable. See the hardware reference guide for the development board.*

5.6.1 Interrupt controller

The interrupt controller is compliant with the *ARM Reference Peripheral Specification* (ARM DDI 0062D).

The interrupt controller provides a simple software interface to the interrupt system.

In an ARM system, two levels of interrupt are available:

- FIQ (Fast Interrupt Request) for fast, low latency interrupt handling
- IRQ (Interrupt Request) for more general interrupts

Ideally, in an ARM system, only a single FIQ source would be in use at any particular time. This provides a true low-latency interrupt, because a single source ensures that the interrupt service routine may be executed directly without the need to determine the source of the interrupt. It also reduces the interrupt latency, because the extra banked registers that are available for FIQ interrupts may be used to maximum efficiency by preventing the need for a context save.

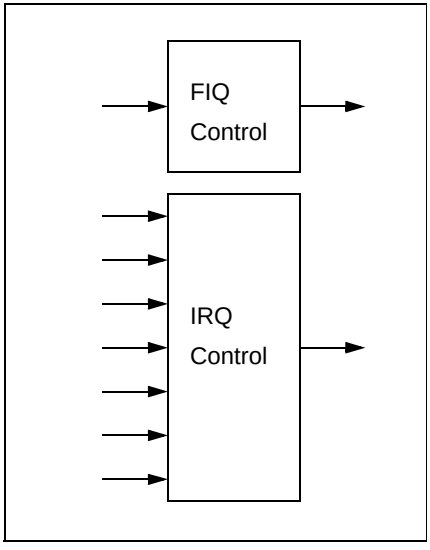


Figure 5-5: FIQ and IRQ interrupts

Separate interrupt controllers are used for FIQ and IRQ. Only a single bit position is defined for FIQ, which is used by a single interrupt source, while 15 bits of the 32 bits available in the IRQ controller are used.

Open Access

The IRQ interrupt controller uses a bit position for each different interrupt source and bit positions are defined for a software programmed interrupt, communication channels, timers, PC Card slots, and for ASB and APB. The bit positions are shown in ***Table 5-6: IRQ bit allocation*** on page 5-17. Bit 0 is unassigned in the IRQ controller so that it may share the same interrupt source as the FIQ controller.

All IRQ interrupt source inputs are active HIGH and level sensitive. Any inversion or latching required to provide edge sensitivity must be provided at the generating source of the interrupt.

A hardware priority scheme is not provided, neither is any form of interrupt vectoring, because these functions can be provided in software.

A programmed interrupt register is provided to generate an interrupt under software control. Typically, this is used to downgrade a FIQ interrupt to a IRQ interrupt.

Memory map

The mapping of the interrupt controller registers conforms to the *ARM Reference Peripheral Specification (RPS)* with a base address of 0x0A000000, giving the map shown in ***Table 5-5: Interrupt controller memory map***.

Address	Read location	Write location
0x0A000000	IRQStatus	Reserved
0x0A000004	IRQRawStatus	Reserved
0x0A000008	IRQEnable	IRQEnableSet
0x0A00000C	Reserved	IRQEnableClear
0x0A000010	Reserved	IRQSoft (Programmed IRQ Interrupt)
0x0A000100	FIQStatus	Reserved
0x0A000104	FIQRawStatus	Reserved
0x0A000108	FIQEnable	FIQEnableSet
0x0A00010C	Reserved	FIQEnableClear
0x0A000110	Reserved	Reserved
0x0A000114	FIQSource	FIQSource

Table 5-5: Interrupt controller memory map

Open Access

Register descriptions

The following registers are provided for both FIQ and IRQ interrupt controllers:

Enable Register	Read-only. The enable register is used to mask the interrupt input sources and defines which active sources generates an interrupt request to the processor. This register is read only and its value can be changed only by the enable set and enable clear locations. If certain bits within the interrupt controller are not implemented, the corresponding bits in the enable register are read as undefined. An enable bit of 1 indicates that the interrupt is enabled and allows an interrupt request to reach the processor. An enable bit of 0 indicates that the interrupt is disabled. On reset, all interrupts are disabled.
Enable Set	Write-only. This location is used to set bits in the interrupt enable register. When writing to this location, each data bit that is high causes the corresponding bit in the enable register to be set. Data bits that are low have no effect on the corresponding bit in the enable register.
Enable Clear	Write-only. This location is used to clear bits in the interrupt enable register. When writing to this register, each data bit that is high causes the corresponding bit in the enable register to be cleared. Data bits that are low have no effect on the corresponding bit in the interrupt enable register.
Source Status	Read-only. This location provides the status of the interrupt sources to the interrupt controller. A high bit indicates that the appropriate interrupt request is active prior to masking.
Interrupt Request	Read-only. This location provides the status of the interrupt sources after masking. A high bit indicates that the interrupt is active and generates an interrupt to the processor.
Programmed IRQ Interrupt	Write-only. A write to this register sets or clears a programmed interrupt. Writing to this register with bit 1 set high generates a programmed interrupt, while writing to it with bit 1 set low clears the programmed interrupt. The value of this register may be determined by reading bit 1 of the Source Status register. Bit 0 of this register is not used.
FIQ Source	The interrupt controller implemented on the ARM Development Board has an extra register, FIQSource, that enables the source of the FIQ interrupt to be selected as any of the 15 IRQ sources (1-15), or an external FIQ source pin.

Open Access

Write 0x0 to this register for the external FIQ source to be selected and 0x1 to 0xF to select the IRQ source corresponding to the bit allocation defined in ***Table 5-6: IRQ bit allocation***.

The external FIQ source is an active low, level-sensitive input on the ASB and APB expansion connectors. This input is pulled HIGH on the ARM Development Board.

A number of registers are reserved for test. These registers must not be accessed during normal operation.

Table 5-6: IRQ bit allocation shows the bit assignment of the IRQ interrupt controller registers 0x0A000000 to 0xA000010.

Bit	Interrupt source
0	Unused
1	Programmed interrupt
2	Debug Channel Comms Rx
3	Debug Channel Comms Tx
4	Timer 1 (internal)
5	Timer 2 (internal)

Table 5-6: IRQ bit allocation

Open Access

Bit	Interrupt source
6	PC Card slot A
7	PC Card slot B
8	Serial port A
9	Serial port B
10	Parallel port
11	ASB expansion 0
12	ASB expansion 1
13	APB expansion 0
14	APB expansion 1
15	APB expansion 2

Table 5-6: IRQ bit allocation

Table 5-7: FIQ bit allocation shows the bit assignment of the FIQ interrupt controller registers 0x0A000100 to 0x000010C.

Bit	Interrupt source
0	External FIQ source
1-15	Unused

Table 5-7: FIQ bit allocation

5.6.2 Timers

Two 16-bit counter timers with 8-bit prescalers are implemented as defined in the *ARM Reference Peripherals Specification* (ARM DDI 0062D).

Each timer is a 16-bit wide down counter with selectable prescale. The prescaler enables either the system clock to be used directly, or the clock divided by 16 or 256 may be used. This is provided by 0, 4, or 8 bits of prescale.

Two modes of operation are available:

- free-running
- periodic

Open Access

In periodic timer mode the counter generates an interrupt at a constant interval. In free-running mode, the timer overflows after reaching its zero value and continues to count down from the maximum value.

The timer is loaded by writing to the Load register and then, if enabled, the timer counts down to zero. On reaching a count of zero, an interrupt is generated. The interrupt may be cleared by writing to the Clear register.

After reaching a zero count, if the timer is operating in free-running mode it continues to decrement from its maximum value. If periodic timer mode is selected, the timer reloads from the Load register and continues to decrement. In this mode the timer effectively generates a periodic interrupt. The mode is selected by a bit in the Control register.

At any point, the current timer value can be read from the Value register.

The timer is enabled by a bit in the control register. At reset, the timer is disabled, the interrupt is cleared, and the Load register is undefined. The mode and prescale value are also undefined.

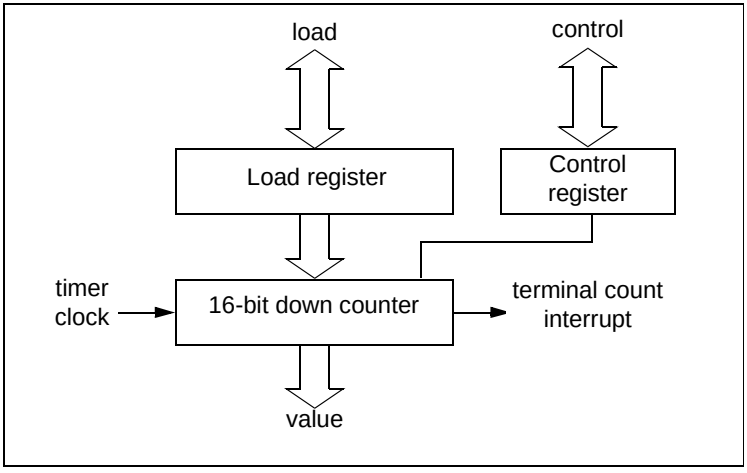


Figure 5-6: Timer block diagram

The timer clock is generated by a prescale unit. The timer clock may be:

- the system clock
- the system clock divided by 16, which is generated by 4 bits of prescale
- the system clock divided by 256, which is generated by a total of 8 bits of prescale

Open Access

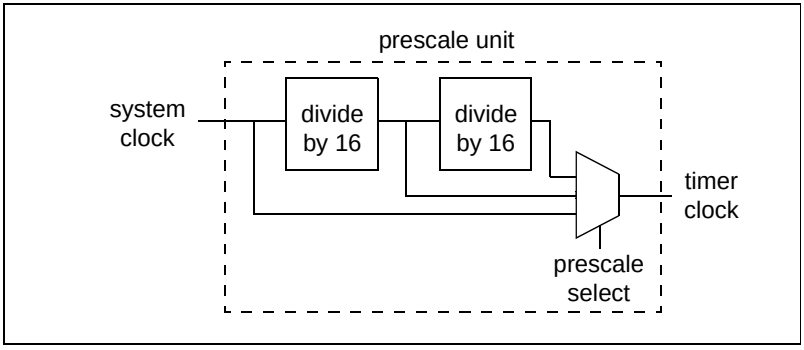


Figure 5-7: Timer prescale unit

Memory map

The mapping of the registers conforms to the RPS with a base address of 0x0A800000, giving the memory map shown in ***Table 5-8: Timer memory map***.

Address	Read location	Write location
0x0A800000	Timer1Load	Timer1Load
0x0A800004	Timer1Value	Reserved
0x0A800008	Timer1Control	Timer1Control
0x0A80000C	Reserved	Timer1Clear
0x0A800020	Timer2Load	Timer2Load
0x0A800024	Timer2Value	Reserved
0x0A800028	Timer2Control	Timer2Control
0x0A80002C	Reserved	Timer2Clear

Table 5-8: Timer memory map

Register descriptions

Load Register — Read-write. The Load register contains the initial value of the timer and is also used as the reload value in periodic timer mode. When writing to this register, the top 16 bits must be written as 0. When reading, the top 16 bits are undefined.

Open Access

Value	Read-only. The Value location gives the current value of the timer. When reading this location, the top 16 bits are read as undefined.
Clear	Write-only. Writing to the Clear location clears an interrupt generated by the counter timer.
Control Register	Read-write. The Control register provides enable/disable, mode and prescale configurations for the timer.

Figure 5-8: Timer Register Bit Positions and ***Table 5-9: Bits 3 – 2, prescale bits*** below define the operation of the Control register.

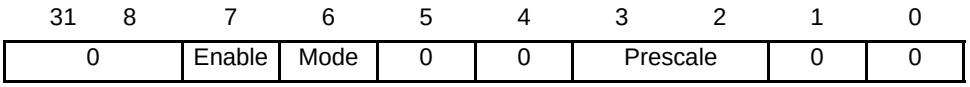


Figure 5-8: Timer Register Bit Positions

Bit 7 – Enable Bit
0—Timer Disabled
1—Timer Enabled

Bit 6 – Mode Bit
0—Free-running Model
1—Periodic Timer Model

Bits 3 – 2:
Prescale Bits
See ***Table 5-9: Bits 3 – 2, prescale bits***

Bit 3	Bit 2	Clock divided by	Stages of prescale
0	0	1	0
0	1	16	4
1	0	256	8
1	1	Undefined	

Table 5-9: Bits 3 – 2, prescale bits

Bits 31 – 8, 5 – 4, 1 – 0:
Undefined
Must be written as zero
Read as undefined

Open Access

5.6.3 Remap and pause

The remap and pause control is the combination of four separate functions:

- pause
- identification register
- reset status
- reset memory map

They are all implemented on the ARM Development Board and are compliant with *ARM Reference Peripherals Specification* (ARM DDI 0062D).

Memory map

The mapping of the Remap and Pause Registers conforms to the Reference Peripheral Specification with a base address of 0x0B000000, giving the memory map shown in ***Table 5-10: Remap and Pause memory map***.

Address	Read location	Write location
0x0B000000	Reserved	Pause
0x0B000010	Identification	Reserved
0x0B000020	Reserved	ClearResetMap
0x0B000030	ResetStatus	ResetStatusSet
0x0B000034	Reserved	ResetStatusClear

Table 5-10: Remap and Pause memory map

Pause

Pause — Write-only. This control simply defines a method of allowing the processor system to enter a low-power, "wait for interrupt" state, in which the system does not require the processor to be active.

Writing to the Pause location causes the system to enter the "wait for interrupt" state. The exact effect of writing to this location is not defined, but typically it would prevent the processor from fetching further instructions until it receives an interrupt.

Identification

Identification (ID) — Read-only. This register provides an indication of the system configuration.

Only a single-bit implementation is required, bit 0, which is used

Open Access

to indicate whether there is any further ID information.

ID Bit 0 is the Identification Bit which may be set to:

- 0—no Further ID Information
- 1—further ID Information is available

If the bottom bit of the ID register is set, further bits are required to provide more detailed system identification information.

Clear Reset Map

Clear Reset Map — Write-only. This location provides a method of overlaying the system base memory at reset. Writing to the Clear Reset Map location causes the system memory map to change to normal operation.

When the reset memory map has been cleared and the normal memory map is in use, there is no method of resuming the reset memory map, other than resetting the board.

Reset Status

Reset Status Register — Read-only. This location indicates the cause of the most recent reset condition—a minimum implementation is defined below.

Only one bit of this register defined, the Power On Reset bit. This bit may be used to determine if the most recent reset was caused by initial power on, or if a warm reset has occurred. The Power On Reset bit is bit 0 of the Reset Status Register, and takes the values:

- 0—no Power On Reset since flag was lasted cleared
- 1—Power On Reset

The Reset Status register has a dual mechanism for setting and clearing bits, allowing independent bits to be changed with no knowledge of the other bits in the register.

Reset Status Clear — Write-only. This location is used to clear Reset Status flags. When writing to this register, each data bit that is high causes the corresponding bit in the Reset Status register to be cleared. Data bits that are low have no effect on the corresponding bit in the Reset Status register.

Reset Status Set — Write-only. This location is used to set Reset Status flags. When writing to this register, each data bit that is high causes the corresponding bit in the Reset Status register to be set. Data bits that are low have no effect on the corresponding bit in the Reset Status register.

This location has no function in the ARM Development Board, because the Power On Reset status bit cannot be set by software. This register is included in the specification to ensure expandability of the reset status functionality.

Open Access

5.7 Serial, Parallel, and PC Card Ports

Two serial ports, one parallel port, and a two slot PC Card (PCMCIA) interface are implemented on the ARM Development Board primarily for communication to a host debugging system, although they are available for other uses too.

Standard PC style peripherals are used and placed on a NISA (not ISA) bus accessible via an ASB/NISA bridge.

A region of memory is reserved for the NISA bridge. The bottom 32MB of the region is I/O space, and the top 32MB is memory space for the PC Card interface.

This region is the same for both Normal and Reset configurations of the memory map, as shown in ***Figure 5-9: NISA region***.

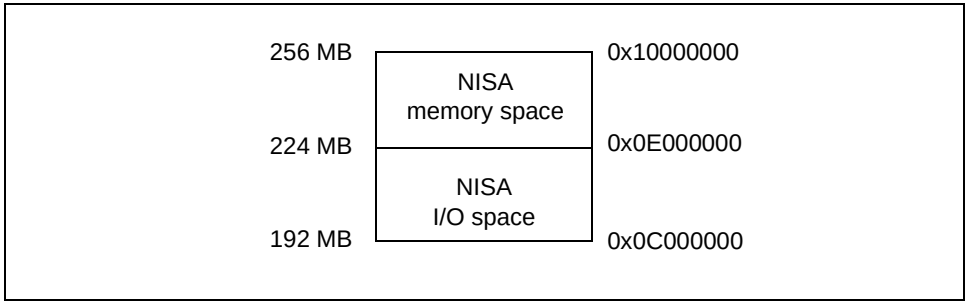


Figure 5-9: NISA region

5.7.1 NISA address map

The base addresses for the communications ports are decoded in the NISA interface and the registers of the communications ports are word-aligned, although only the lower 8 bits are used to transfer data. The communications ports are located in the top 8MB of I/O space, and the address decoding is such that they are aliased over the whole 8MB.

The PCMCIA controller is self-decoding, and is controlled through two 8-bit registers, which are accessed using byte transfers. The PCMCIA memory and I/O space exists as16-bit data aligned on 32-bit boundaries, which means that address bit A0 selects the upper or lower byte, but bit A1 is unused.

The address map of the NISA space is shown in ***Table 5-11: Address map of the NISA space***.

Open Access

Addresses	Description
0x0C0007C0 to 0x0C0007C1	PCMCIA control registers
0x0D800000 to 0x0D80001F	Serial port A
0x0D800020 to 0x0D80003F	Serial port B
0x0D800040 to 0x0D80005F	Parallel port
0x0D800060 to 0x0DFFFFFF	Aliased images of comms ports
0x0E000000 to 0x0FFFFFFF	PCMCIA memory

Table 5-11: Address map of the NISA space

5.7.2 Serial and parallel ports

The serial and parallel I/O subsystem is based on an ST16C552 device. This is a dual asynchronous receiver and transmitter with 16-byte transmit and receive FIFO and a bidirectional Centronics type parallel printer port. This device is pin compatible and functionally compatible with the VL16C552 and the WD16C552. See ***Appendix A, ST16C552 UART with FIFO and Parallel Printer Port*** for a data sheet for programming this device.

Address offsets for the serial port registers are shown in ***Table 5-12: Addresses for the serial port registers*** on page 5-26.

Open Access

Serial Port A	Serial Port B	Read mode	Write mode
0x0D800000[A]	0x0D800020	Rx Holding	Tx Holding
0x0D800004	0x0D800024		Interrupt Enable
0x0D800008	0x0D800028	Interrupt Status	FIFO Control
0x0D80000C	0x0D80002C	Line Control	
0x0D800010	0x0D800030		Modem Control
0x0D800014	0x0D800034	Line Status	
0x0D800018	0x0D800038	Modem Status	
0x0D80001C	0x0D80003C	Scratchpad	Scratchpad
0x0D800000[A]	0x0D800020[A]		Divisor Latch LSB
0x0D800004[A]	0x0D800024[A]		Divisor Latch MSB

Table 5-12: Addresses for the serial port registers

A The registers are enabled in baud rate setup mode. Note that this mode must be disabled for normal operation. See the chip set data sheet for details in ***Appendix A, ST16C552 UART with FIFO and Parallel Printer Port***.

Address offsets for the parallel port registers are shown in ***Table 5-13: Addresses for the parallel port registers***.

Address	Read mode	Write mode
0x0D800040	Data	Data
0x0D800044	Status	I/O Select
0x0D800048	Command	Control

Table 5-13: Addresses for the parallel port registers

All register definitions are included in the `nisa.h` include files that are in the target development system example software suite.

Open Access

Some features have been added to the board to enable the parallel port to drive some LEDs and read switches.

For information about how to configure the ARM Development Board to use these features, see ***6.6 Parallel Port/LED mode configuration*** on page 6-9.

Note *When you use the parallel port to connect to external devices, you are advised to remove all the jumpers in the link field (LK11) and the link (LK10).*

5.7.3 PC Card Interface

The PC Card (PCMCIA) I/O is based on a Vadem VG-468 (U25) device. This is a PC Card socket controller that is designed to connect to a PC ISA bus. With the ARM Development Board, it is driven by the NISA bridge. The controller supports two PC Card sockets and the associated voltage switching devices. The two socket connectors form one physical unit, housing two card slots. The upper slot is A, and the lower slot is B.

To program the controller, you are advised to obtain a data sheet from the manufacturer. See www.vadem.com for your local distributor. Contact ARM if you have any problems obtaining a data sheet.

The ARM Development Board supports PC Cards that require a voltage (VCC) of +5V, and a programming voltage (VPP) of +5V.

Note *No support is provided for +3.3V PC Cards.*

Power supply configuration

The PC Card supply voltage (VCC) is controlled by the **nVCCEN** signal from the Vadem VG-468 device. Setting this signal LOW sets the card VCC voltage to +5V. Setting it HIGH puts the power supply pins into high impedance, effectively cutting off the power supply to the card. This is summarized in ***Table 5-14: VCC supply***.

nVCCEN	VCC supply
0	+5V
1	HIGH Z

Table 5-14: VCC supply

The PC Card programming voltage (VPP) is controlled by two signals, **VPP1EN** and **VPP2EN**, from the Vadem VG-468 device. By setting these signals HIGH or LOW, the supply can be switched between +5V, +12V, GND, and high impedance, as shown in ***Table 5-15: VPP supply***.

Open Access

VPP1EN	VPP2EN	VPP supply
0	0	0V
0	1	+5V
1	0	+12V
1	1	HIGH Z

Table 5-15: VPP supply

LEDs

There are two yellow LEDs (D11 and D12) attached to the VG-468, labeled PCA and PCB, associated with PC Card sockets A and B, respectively. These LEDs are connected to the DGIO pins of the controller, and so can be switched on and off by writing to an appropriate register in the device.

Address	Read mode	Write mode
0x0C0007C0		Control
0x0C0007C1	Data Read	Data Write

Table 5-16: PC Card control registers

The PC Cards are operated by writing the required register value to the Control register, and reading from or writing to the Data register.

A comprehensive include file for the PC Card chip set, `pcmcia.h`, is included with the target development system example software suite.

For more information about PC Card data transfer, see ***7.6.2 Accessing the Vadem PCMCIA controller*** on page 7-16.

Open Access

6 Configuring the ARM Development Board

6.1	Introduction	6-2
6.2	Setting the Main System Clock Frequency	6-4
6.3	Remapping the Memory	6-5
6.4	Setting the SRAM Speed and Width	6-6
6.5	Setting the EPROM/Flash Speed and Width	6-7
6.6	Parallel Port/LED mode configuration	6-9
6.7	Big-endian Operation	6-12

Open Access

Configuring the ARM Development Board

6.1 Introduction

This chapter gives a brief overview of the ARM Development Board, and details of how to configure the board to emulate different systems.

The ARM Development Board system comprises:

- the processor daughter board
- the peripheral mother board
- a parallel port adaptor

6.1.1 The processor daughter board

On the 7TDMI version, the daughter board contains the ARM7TDMI core processor and the necessary logic to make it an AMBA bus master. The processor and all daughter board interface signals run at 3.3V.

6.1.2 The peripheral mother board

This contains many of the elements used in ASIC and microcontroller applications.

When the board is shipped, it is set in the following state:

- the main system clock (**B_CLK**) is set to 20MHz
- single cycle 32K x 32-bit Synchronous Static RAM
- 512K x 32-bit SRAM set up as 2 banks of 32-bit 2-cycle wait state memory
- 512k x 8-bit Flash EPROM
- no DRAM
- parallel port
- two serial ports
- no PC Cards

You may want to change the default board setup to emulate different systems and to observe the effects of different peripherals on system performance.

6.1.3 Parallel port adaptor

This board is included primarily to enable the parallel port to be used to communicate with a host computer's parallel port (rather than being used with peripherals, such as a printer).

Open Access

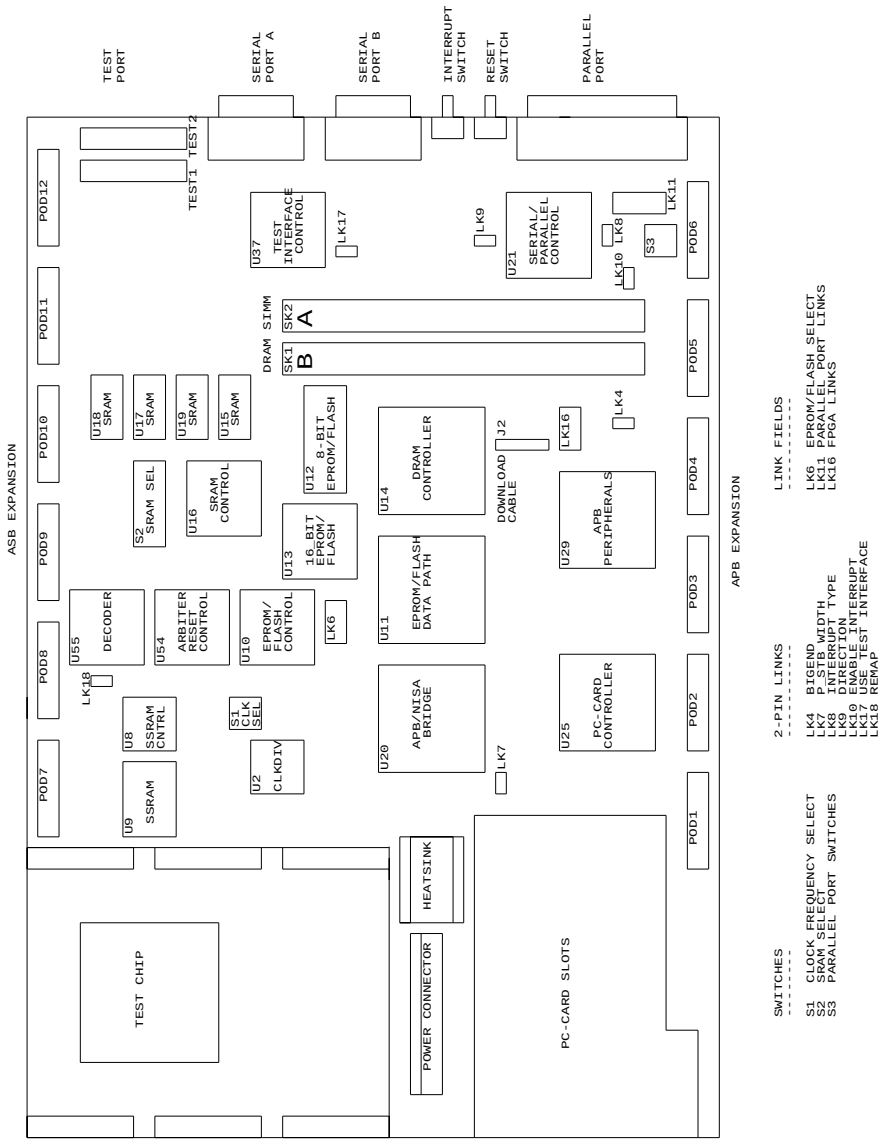


Figure 6-1: ARM Development Board Component Layout

Open Access

6.2 Setting the Main System Clock Frequency

This is set by the DIP switch (S1) labeled FREQ SELECT on the top left-hand side of the mother board, close to the daughter board. Select the frequency by setting the switches according to ***Table 6-1: System Clock settings***.

Position1	Position2	Position3	Position4	SYSCLK (MHz)
On	On	On	On	4
Off	On	On	On	8
On	Off	On	On	16
Off	Off	On	On	20

Table 6-1: System Clock settings

Note *Do not select any combination other than that shown in the table.*

Open Access

6.3 Remapping the Memory

The address decoder functions in two distinct states: Normal and Reset. At power up, by default the board is in the Reset configuration. In this state, the EPROM or Flash can be found at the bottom of the address map. If the remap register is written to, the **REMAP** signal goes HIGH, and the decoder switches to the normal memory map in which there is SRAM at the bottom.

If you are bringing up a system in which there is no EPROM or Flash, you may want to disable this feature. Link (LK18) is provided for this purpose. In the default position (in), the **REMAP** input to the decoder is driven by the remap and pause controller implemented in the APB FPGA. If this link is removed, the **REMAP** signal is always HIGH, and the board starts up with SRAM at the bottom of the address map.

Ref	Name	Description	Options	Default
LK18	REMAP	driven or always HIGH	out = HIGH in = driven	in

Table 6-2: Memory remap with LK18

For a more detailed description of the memory map and the remapping feature, see ***5.3 Main Memory Map*** on page 5-7.

Open Access

6.4 Setting the SRAM Speed and Width

The static RAM is configured in two banks of equal size. These can each be defined independently to allow emulation of applications using two different types of memory (such as 16-bit fast RAM with 8-bit slow EPROM).

The relative speed of the memory is configured by inserting extra cycles into memory accesses. To do this, set switch S2 to one of the positions from ***Table 6-3: SRAM speed settings on S2***.

Bank0		Bank1		Memory cycles
Position1	**Position2**	**Position5**	**Position6**	
On	On	On	On	2
Off	On	Off	On	3
On	Off	On	Off	4
Off	Off	Off	Off	5

Table 6-3: SRAM speed settings on S2

You control the data bus size by setting switches on S2 as shown in ***Table 6-4: SRAM Data Bus Width Settings on S2***.

Bank0		Bank1		Bus size
Position3	**Position4**	**Position7**	**Position8**	
On	On	On	On	8-bits
Off	On	Off	On	16-bits
On	Off	On	Off	32-bits
Off	Off	Off	Off	32-bits

Table 6-4: SRAM Data Bus Width Settings on S2

Open Access

6.5 Setting the EPROM/Flash Speed and Width

You configure the EPROM/Flash memory speed and data bus width using the link field (LK6) labeled EPROM SELECT at the center of the board. These are numbered from the right-hand side, as shown in ***Figure 6-2: EPROM SELECT link field (LK6) pinout***.

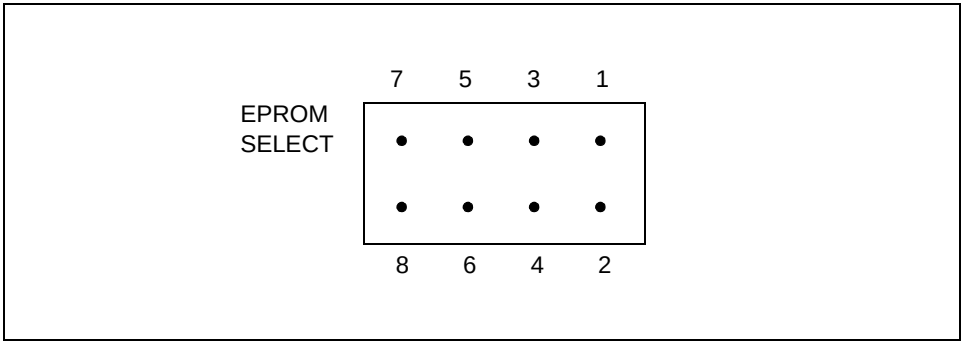


Figure 6-2: EPROM SELECT link field (LK6) pinout

Link positions 1-2 and 3-4 control the memory speed, changing the widths of the **CE**, **OE** and **WE** signals. The widths in ***Table 6-5: EPROM/Flash speed settings on LK6*** are calculated for a system running at **MCLK** = 20MHz (cycle time = 50ns).

Position 1-2	Position 3-4	Cycles	CE (ns)	OE (ns)	WE (ns)
In	In	2	100	100	50
In	Out	3	150	150	100
Out	In	4	200	200	150
Out	Out	5	250	250	200

Table 6-5: EPROM/Flash speed settings on LK6

Open Access

Flash memory or conventional EPROM can be used, selected with position 5-6, as shown in ***Table 6-6: EPROM/Flash memory type setting on LK6***.

Position 5-6	Memory type
In	Flash
Out	EPROM

Table 6-6: EPROM/Flash memory type setting on LK6

8-bit devices can be used in socket U12 or 16-bit devices in socket U13, as shown in ***Table 6-7: EPROM/Flash width setting on LK6***.

Position 7-8	Bus width
In	16-bit
Out	8-bit

Table 6-7: EPROM/Flash width setting on LK6

Open Access

6.6 Parallel Port/LED mode configuration

The parallel port can either be configured as a standard communication port or as a special LED port. The configuration is controlled by the link field (LK11), which is in the bottom right-hand corner of the board (beside the parallel port connector and below the 16C552 UART). See ***Table 6-8: LED and switch configuration on LK11***.

In LED mode, the lower nibble of the parallel port is mapped to switch S3, and the upper nibble to LEDs D7-10. In this way, you can read the switches and write to the LEDs using the parallel port. This mode is selected if all eight links are in (connected).

Standard parallel communication port connections to an external device are selected by removing all eight of the links (not connected). See ***2.5.3 Serial/parallel connection*** on page 2-22 for information about how to connect the parallel port to your host computer.

Ref	Position	Options
LK11	1	out = use parallel port in = PP bit0 from S3-1
	2	out = use parallel port in = PP bit1 from S3-2
	3	out = use parallel port in = PP bit2 from S3-3
	4	out = use parallel port in = PP bit3 from S3-4
	5	out = use parallel port in = PP bit4 to LED PP0
	6	out = use parallel port in = PP bit5 to LED PP1
	7	out = use parallel port in = PP bit6 to LED PP2
	8	out = use parallel port in = PP bit7 to LED PP3

Table 6-8: LED and switch configuration on LK11

Note *If you need to drive some LEDS and use an external device, you should consider driving the LEDs connected to the PC Card (PCMCIA) controller. Some example code is described in* ***7.5 Simple PC Card (PCMCIA) Access*** on page 7-12.

Open Access

The ST16C552 can be configured using two links, INT TYPE (LK8) and DIRN (LK9). LK8 selects the interrupt type, either latched mode (out) or ACK mode (in). In “latched mode” (default), a falling edge on the ACK pin is latched and causes a parallel port interrupt. This interrupt is cleared by reading the appropriate status register. In “ACK mode”, the ACK pin is connected directly to the parallel port interrupt line.

The DIRN link (LK9) specifies whether the parallel port is bidirectional (out) or output only (in). The default is bidirectional.

If the parallel port ACK link is not connected externally, it is possible to cause **ACK** to pulse LOW by pressing the momentary action switch (SW1) labeled INTERRUPT, which has a black cap. To enable this function, the link ENABLE INT (LK10) must be in. If the link is out, pressing the switch has no effect.

Note *When you use the parallel port to connect to external devices, you are advised to remove the jumper from ENABLE INT (LK10).*

The three links are summarized in ***Table 6-9: Interrupt and direction configuration***.

Ref	Name	Description	Options
LK8	INT TYPE	latched or ACK mode	out = latched in = ACK
LK9	DIRN	parallel port direction	out = i/o in = output
LK10	ENABLE INT	enable switch interrupt	out = disabled in = enabled

Table 6-9: Interrupt and direction configuration

Configuring the parallel port adaptor

The parallel port adaptor has two modes of operation. These modes are selected by the link (LK1). When the link is in place, the adaptor is configured for connection to a host PC (for example, for download). When the link is removed, the adaptor is configured to appear as a standard PC parallel port with the correct pinout. Thus, a printer or similar peripheral could be driven from the female connector (J2).

Open Access

Link state	Mode
In	Peripheral (download)
Out	Host

Table 6-10: Download configuration LK1

Open Access

6.7 Big-endian Operation

The ARM Development Board is designed to operate byte order independently with very minor modification.

For board operation with RAM resident code, (all examples except the selftest and stand alone), board endian operation is controlled by the BIGEND link (LK4). When this link is in place, the board operates as a big-endian system. For big-endian operation, you download code generated using the big-endian option on the assembler or compiler. With the link out, the board mode is little-endian.

If ROM accesses are required, the ROM controller chip (U10) must be replaced. This is labeled EFI-0019 and is situated above the center of the main board, immediately above the ROM setup links (LK6). The board is supplied with the little-endian chip fitted as standard. You can obtain the big-endian chip from ARM (see www.arm.com), identification number (EFI-0084).

To operate correctly in big-endian mode, the BIGEND link (LK4) must be fitted when operating ROM-based code.

Note *If you are using only the EmbeddedICE interface and you are not running code from EPROM/Flash, you do not need to replace the ROM controller chip.*

6.7.1 Preparing code for big-endian operation

Windows operation

To configure the ARM Project Manager for big-endian operation on all projects:

1. Select **Configure→cc** from the Tools menu.
2. Click on the **Target Configuration** tab.
3. Select the **Big endian** option.
4. Click on **OK**.
5. Repeat steps 1 to 4 for **Configure→asm**.

To configure the ARM Debugger for Windows:

1. Select **Configure Debugger** from the Options menu.
2. Click on the **Debugger** tab.
3. Click on **Big** in the Endian section.
4. Click on **OK**.

You can then use the project manager and debugger in the usual way.

Open Access

Command-line operation

Use the example described in ***Chapter 3, A Worked Example*** with the following changes:

When you compile, the command-line option for either armcc or tcc is:

```
tcc leds.c -bi -i ..\include -o leds
```

where `-bi` is the big-endian switch for the compiler.

Note *If you are using armasm for any code, you must set the* `-bi` *flag.*

To run the resultant code on the ARM Development Board, insert the BIGEND link (LK4) to change the operation of the board RAM to big-endian. Then start the debugger in big-endian mode with the following command:

```
armsd -bi -port s -linespeed 38400 leds
```

The debugger replies with:

```
A.R.M. Source-level Debugger vsn 4.46 (Advanced RISC Machines SDT2.10)
       [Dec 13 1996]
EmbeddedICE Manager (ADP ARM7TDMI) 2.01 (Advanced RISC Machines)
Object program file leds
armsd:
```

Type g at the prompt to execute the program.

6.7.2 EPROM memory organization

This section describes how the EPROM controller stores code in memory.

Figure 6-3: Example ARM instruction AABBCCDD shows the assumed code format for an example instruction AABBCCDD.

Open Access

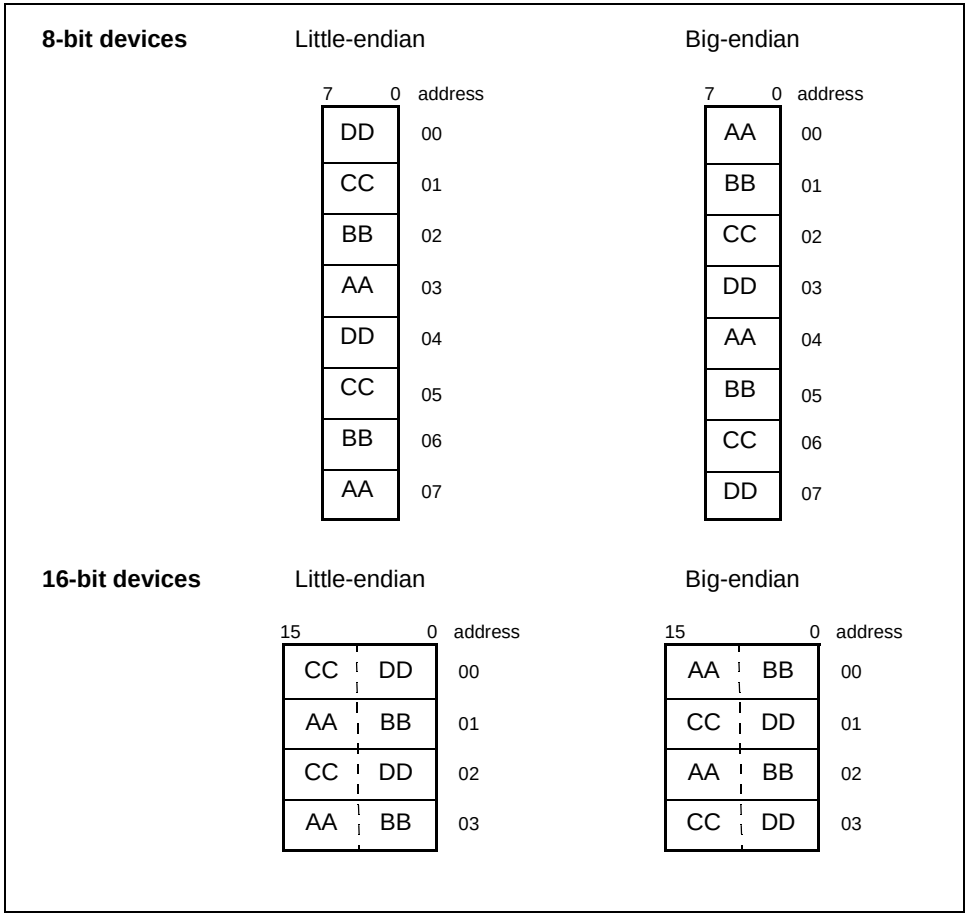


Figure 6-3: Example ARM instruction AABBCCDD

If you use the Flash download program you do not need to worry about how the code is stored in memory. However, if you use an EPROM programmer to program the device, this may be important. In general, programming little-endian code into 8- and 16-bit devices, or big-endian code into 8-bit devices does not cause any problems. If you want to program big-endian code into 16-bit devices using an EPROM programmer, you may need to swap even and odd bytes to get the device to program as required. Most EPROM programmers allow you to do this.

To check, use EmbeddedICE to examine the memory and verify that instructions are being read correctly.

Open Access

7 Example Code Tutorials

7.1	The Example Code Suite	7-2
7.2	Driving the Parallel Port	7-4
7.3	Driving the Serial Port	7-6
7.4	A Simple Interrupt Handler Routine	7-9
7.5	Simple PC Card (PCMCIA) Access	7-12
7.6	PC Card (PCMCIA) SRAM Memory Card Read/Write Example	7-14
7.7	A ROM Resident Program	7-18
7.8	Flash Downloader	7-24

Open Access

Example Code Tutorials

7.1 The Example Code Suite

This suite of software routines has been written to show practical examples of writing code for the 7TDMI version of the ARM Development Board. The aim is to demonstrate how you set up and access some of the peripheral systems on the board by using example source code that can be compiled, downloaded, and run to observe its operation.

Note *If you are using Angel, you must take into account additional considerations before you compile code. These are documented in Application Note 43:* Design Considerations when using Angel Debug Monitors *(ARM DAI 0043). You can then compile the "leds.c" example described in this chapter to run on Angel.*

All the examples are written as stand-alone files, using the standard C libraries for compilation. Most files are contained in a single source, though some use system 'include' files for ease of compilation and understanding.

The suite contains a set of C source files, include files and binary images of board utilities.

7.1.1 C source files

`leds.c`	The parallel port has four LEDs and four switches linked to the upper and lower nibbles, respectively. This code demonstrates how to access the port, reads the switches, and uses the read value as a seed for the frequency of an LED shifter.
`serial.c`	Shows how to use the two on-board serial ports in test mode, and how to use them operating as a link to transmit and receive simple strings.
`pcmc.c`	Operates the PC Card registers at a very basic level to toggle two general purpose bits in the PC Card chip that are connected to LEDs on the board.
`pcmcsram.c`	Requires a PC Card RAM card. Reads-writes memory locations and displays their contents.
`interrupt.c`	Shows the operation of a simple interrupt handler written in C by using a counter timer interrupt to toggle the board's LEDs.
`dhry_1pt.c`, `dhry_2pt.c`, `timerpt.c`	Shows how to modify the standard Dhrystone throughput test code for operation on the ARM Development Board. Differences between Thumb-state and ARM-state code, and operation with various memory setups are discussed.

Open Access

`boot.s`, `stand_al.c`	These files generate a ROM code example which can be left resident in the Flash EPROM, and which runs from power up or system reset. The code flashes the LEDs at different speeds and in differing sequences depending upon the setting of the parallel port switches.

7.1.2 Include files

`stdio`, `stdlib`	The standard C library calls to allow the code to be compiled and run stand-alone without any initialization code.
`adb.h`	Defines all of the interrupt and specific ARM Development Board base addresses. This file also calls the `rps.h` include file.
`rps.h`	Defines all of the peripheral base addresses as defined in the *ARM Reference Peripheral Specification (RPS)* (ARM DDI 0062D).
`st16c552.h`	Defines the addresses of the registers used in the ST16c552 Serial/Parallel interface chip.
`pcmcia.h`	Defines the register base addresses of the Vadem chipset used to control the dual PC Card slots on the ARM Development Board.
`dhry.h`, `time.h`	Define the clock source and general defines for the Dhrystone throughput test explained in ***Chapter 8, Benchmarking on the ARM Development Board***.
`stand.h`	Contains miscellaneous definitions for the stand-alone ROM LED Sequencer.

7.1.3 Board utilities

These files are used to ensure that the board is in an operational condition:

`selftest`	Initial self-test of many of the board functions is performed. The image file can be downloaded using the Flash download facility described in ***2.7 Using the Flash Downloader*** on page 2-31.
`diagnose`	This file must be downloaded to the board using an EmbeddedICE interface. It carries out testing to a greater level than selftest and reports its findings to the host computer.

Open Access

7.2 Driving the Parallel Port

This routine shows how to read and write to the parallel port on the ARM Development Board. On the board, a series of links is used to connect switches and LEDs to the parallel port. These switches and LEDs are used as the inputs/outputs to demonstrate parallel port operation.

The routine reads the switches and uses the value as a delay element in a loop that writes a shifted single bit to the LEDs.

Note *To allow this routine to operate, there must be no active connection to the parallel port. All connections on Link LK11 must be made.*

7.2.1 leds.c

Include files

`stdio.h`, `stdlib.h`	The standard C libraries have been included to enable the example code to be compiled as a stand-alone program.
`nisa.h`, `adb.h`, `rps.h`	These includes define the required offset values for the peripherals and standard address definitions of the ARM Development Board.
`st16c552.h`	A file containing the register definitions and standard set up values for the 16c552 serial/parallel interface chip.

Local definitions

The defined local values are specific to leds.c. They include the delay elements, the LED offset in the port (pointer to the upper nibble), and the maximum and minimum bit range for the LEDs.

Parallel demonstration

The Parallel Data Registers are initialized to zero to ensure that they contain no spurious data, especially on the upper nibble of the LEDs. This ensures that all LEDs are off at start up.

The LED Shift value is then initialized to 1 to ensure that there is an LED lit and there is some data to be shifted in the loop. A 0 must be written to the switch bits to avoid clashing with the switches which may be tied to ground or Vcc through a resistor.

During the infinite loop, the parallel port control bits are set by reading the port command register, setting the required bit (in this case changing to input or output), then writing the result back to the Control Register. This ensures that a change does not affect any other control bits.

Open Access

The loop initially sets the port into input mode and reads the switch value from the lower nibble. The port is then set into output mode and the LED bit pattern is written to the upper nibble of the parallel port. This makes the LEDs light. Next, the bit pattern is checked to see if the last LED has been reached, in which case the pattern is reset to the first LED.

A delay loop is then executed using a value derived from the length of the square of the switch value. When the delay loop finishes, the loop returns to the start.

main

This is the entry point for the program and is required by the C library. `main` calls the demonstration module and then terminates the program.

Notes

This example shows a very basic way of handling data from the parallel port without the use of interrupts. This method could only be of use in applications in which data passage is not dependent on timing considerations.

Open Access

7.3 Driving the Serial Port

Note *If you are using Angel configured in either serial-only mode, or in serial/parallel mode, you cannot use the serial driver example code. This is because in these modes Angel uses one serial port from the target development system.*

This section of code shows how to access the ARM Development Board's two serial ports using a C program. The code is an example of communication without interrupts and as such remains in loops until the transmit/receive transaction is complete.

The module shows the serial port in two modes of operation:

Loop Back or Self Test Mode	The ports are accessed independently of any external link. Data is written to the Transmit Holding Registers (THR) and is then passed internally to the same port's Receive Holding Registers. No data is passed to the external 'physical' port. This mode is a reliable test for any serial communications program as there are no physical constraints on the transfer.
Twin Port Communication	The two serial ports are connected together by a cable with a suitable Tx - Rx Twist. Communication is carried out between the two ports; one sending, the other receiving data as sent.

7.3.1 serial.c

Include files

`stdio.h`, `stdlib.h`	The standard C libraries have been included to enable the example code to be compiled as a stand-alone program.
`nisa.h`, `adb.h`, `rps.h`	These includes define the required offset values for the peripherals and standard address definitions of the ARM Development Board.
`st16c552.h`	A file containing the register definitions and standard setup values for the 16c552 serial/parallel interface chip.

Local definitions

The defined local values are specific to serial.c.

The test pattern is a simple alternating bit pattern used to ensure that the data byte in the Loop Back test exercises every bit.

The state flags are bits to be set in the controlling byte of the communications loop. Each bit set signifies the end of a transmit or receive operation, usually when an end of data null is sensed.

Open Access

Module routine explanations

TxFifoFull	This routine reads the Transmit Holding Register Full bit (bit 6) of the specified Line Status Register (LSR). It returns a True value if the bit is clear indicating an empty register ready for new data for transmission. A false (zero) value is returned if the bit is set indicating data being held for transmission.
RxFifoEmpty	The receiver equivalent of the above. Here the specified LSR is checked for the Receive Data Ready bit (bit 0). A true value is returned if the bit is clear indicating no received data being held. A false (Zero) value is returned if the bit is set indicating received data being held in the Rx Holding Register.
Serial_Demonstration	This module contains all the code to demonstrate the two methods of serial operation: • Loop Back • standard comms

Loopback operation

Loopback mode is a method of testing the operation of the port without having any external connections available. Data is transmitted to the transmit holding register and is then internally shifted across the port to the receive register ready for reading back.

The routine initializes the port prior to operation. Initiation comprises of resetting the Transmit and Receive FIFO's and enabling their operation by setting the bits in the FIFO Control Register (FCR).

The communications parameters are set up as follows:

The baud rate is set up internally to the chip by enabling the clock divisor latch (setting bit 7 on the Line Control Register - LCR) and loading the divider latch with the desired baud value. In this case 50 baud is loaded into the divider latch registers.

The Line Control Register is then loaded with the word length and stop bit pattern. In this example, 8 bit word length with 1 stop bit is chosen.

Note *The divisor latch enable bit must only be set while the divisor value is written to the register and must be immediately cleared prior to operation of the serial comms. This is because the divisor latches mirror the local addresses of the Rx and Tx holding registers. Failure to observe this bit stops the port from transmitting and receiving data.*

The port is now finally set into Loopback Mode and is ready for data transfer. The transmitted data byte is written to the Transmit Holding Register (THR) after ensuring that the register is clear using the TxFifoFull test discussed above.

The Receive Holding Register is then monitored using the RxFifoEmpty routine until a character is received. The received character is retrieved from the Receive Holding Register (RHR) and the Transmitted and Received bytes are displayed to show that the data has passed through the port unchanged.

This is then repeated for the second serial port, Port B.

Open Access

Twin port communication

The second part of this routine requires a physical link between Port A and Port B, incorporating a twist in the Tx and Rx lines. This link can be left in place during loopback operation.

To communicate successfully, both ports must be initialized to operate in identical modes. The initialization is for 9600 Baud, 8 bit words with 1 stop bit and Loopback mode disabled.

The system is then placed into a loop to service the data transfer between the ports. The loop is terminated by the control flag having all four lower bits set. This happens when each transfer is completed.

The transfer method between the two ports is identical, so only Port A is described.

The communications loop is a serial loop that attempts to transmit a character, then checks for any received data.

First, the transmit string pointer is checked to ensure that the end of string has not been reached.

The current character byte is then written to the THR if it is empty, and the string pointer is incremented to the next character. When the end of string is reached, a null is written to the THR and the TxA Complete bit is set.

The RHR is monitored to check for incoming data. When data is flagged, it is read and stored in the port string buffer.

When a null character is received, it is stored in the buffer to terminate the received string and the RxA Complete bit is set in the loop control flag.

When all four operations are complete, the two received strings are printed to show that transfer has successfully taken place.

main

This is the entry point for the program and is required by the C library. It calls the Demonstration module and then terminates the program.

Notes

This demonstration shows a simple, stand-alone communications routine for data transfer using the serial ports. In a real application, the serial comms would be controlled by interrupts to allow other functions to be carried out while the serial ports are idle. The method of operation of the serial ports shown here could be used in a serial interrupt handler, if required.

Open Access

7.4 A Simple Interrupt Handler Routine

Note *As a stand-alone routine, this code must be compiled using the armcc compiler to produce 32-bit ARM code. The interrupt handler IRQHandler does not function correctly in 16-bit Thumb state. This, and other compilation issues are discussed in* ***Notes*** on page 7-11.

This code demonstrates a method of constructing a simple interrupt handler in C. For this example, the handler has been written for a single timer interrupt source. Interrupt handling is discussed in much greater detail in the *ARM Software Development Toolkit User Guide* (ARM DUI 0040A).

The interrupt is generated by a Counter/Timer which is set up for periodic operation by loading it with the maximum count to give the maximum delay period. The interrupt handler lights the LEDs when link 11 is made.

7.4.1 interrupt.c

Include files

`stdio.h`, `stdlib.h`	The standard C libraries have been included to enable the example code to be compiled as a stand-alone program.
`nisa.h`, `adb.h`, `rps.h`	These includes define the required offset values for the peripherals and standard address definitions of the ARM Development Board.
`st16c552.h`	A file containing the register definitions and standard setup values for the 16c552 serial/parallel interface chip.

Local definitions

The defined local values are specific to `interrupt.c`.

The IRQ vector address is defined here together with the offsets required to calculate the branch value for the new IRQ vector.

To combine a branch address with a branch instruction, the calculated branch address must only be 24-bits long. Masks are defined here to ensure this, and the Branch op-code is defined.

7.4.2 Module routine explanations

Install_Handler

This routine is as defined in the *ARM Software Development Toolkit User Guide* (ARM DUI 0040A), which discusses the Branch Method of installing Handlers from C.

A new vector is calculated by taking the entry address of the interrupt handler (this is passed to the routine), subtracting the address of the vector (passed to the routine), and subtracting an offset to allow for pipelining.

Open Access

This gives a branch address value from the relevant vector to the handler. The branch address is ORed with the branching BAL Op Code (0xea000000) and stored in the interrupt vector space.

The old vector is then passed to the calling routine in case it is required to be reinstalled later.

IRQHandler

This is defined as an interrupt handler by the __irq type definition before its name.

Initially, IRQStatus word is checked to find the source of the interrupt and, in this case, the Timer1 interrupt is filtered out for handling.

When a Timer1 interrupt is sensed, the handler immediately resets the interrupt by writing data (content does not matter) to the Timer1 Clear Register.

Code is then set up in the interrupt handler to write to the parallel port LEDs toggling them on or off.

This is an example of a very simple interrupt handler. Usually, handlers service all interrupt sources using either an `if...then...else` or a `switch` construct. A flag is set to show that an interrupt has been set, control then returning to the main program's interrupt handling code. This minimizes the time that interrupts need be disabled and allows interrupts to be serviced as quickly and efficiently as possible.

Init_Timer1

This routine disables interrupts and then initializes Counter Timer. The Counter Timer is disabled (by clearing the Control Register) and then cleared of any timer interrupts by writing data to the Clear Register.

Initialise_System

This routine only initializes Timer1 since this is the only portion of timer system being used in the example,

The timer and interrupts are initialized by calling Init_Timer1. Then only the IRQ vector Interrupt Handler is installed as described above; other vectors are not used, remain unchanged and have no handler routines.

Set_Timer_And_Loop

This routine demonstrates a method of setting up Timer 1 for operation as a periodic timer with a period of 0.67 seconds. The 50ns system clock counts down from 0xffff to zero with a 256 prescale, giving:

50 ns * 0xffff * 256 = 0.8388s

between period interrupts.

The counter is initialized by writing the count value (0xffff) to the Timer1 Load register, and then sending a byte the Control Register to put the Timer in periodic mode with maximum prescale.

Open Access

Finally, system interrupts are enabled and the code is forced into an infinite loop to allow observation of the interrupt without interference from any other code.

Notes

The most important aspect of this code sequence is the operation of the handler in a 16-bit Thumb system. The exception handler IRQHandler code can not be compiled using the Thumb compiler tcc since the code generated will not handle the return code correctly. The exception handlers, if written in C, *must* be compiled using the armcc compiler.

The exception handlers can be placed in a separate file and compiled separately as the only ARM code in an otherwise Thumb system. This has the benefit of reducing code size.

It is usual for the interrupt handler to be written in assembler for maximum performance. However, there are some instances where a simple C solution is sufficient. This example has shown one of two methods for setting up an IRQ handler routine in C code. A more detailed discussion of ARM exception handling is covered in the *ARM Software Development Toolkit User Guide* (ARM DUI 0040A).

Open Access

7.5 Simple PC Card (PCMCIA) Access

The ARM Development Board has an on-board PC Card (PCMCIA) interface with two available slots. This example shows how to set up the interface and access the general purpose bit for each channel.

7.5.1 pcmc.c

Include files

`stdio.h`, `stdlib.h`	The standard C libraries have been included to enable the example code to be compiled as a stand-alone program.
`nisa.h`, `adb.h` `rps.h`	These includes define the required offset values for the peripherals and standard address definitions of the ARM Development Board.
`st16c552.h`	A file containing the register definitions and standard setup values for the 16c552 serial/parallel interface chip.

Local definitions

The defined local values are specific to pcmc.c.

Two loop maximum values are defined to give a finite length to the LED flashing loop to show the exit requirements of the Vadem chipset.

An offset is defined which is used for conversion of data between the Vadem chipset and the ARM on reads. This procedure is explained in the notes section.

7.5.2 Module routine explanations

The Vadem chip-set registers are accessed by setting the index register to the required register offset and then reading or writing from the register. This offset is then latched so that any further reads or writes can be achieved without setting up the index again.

pccard_regr

The required register offset is passed to this function and is loaded into the index register. This register is then read and the value shifted from the Vadem bus output position to the byte read position required by the ARM. This is necessary because of an inconsistency between the way the Vadem chipset passes read data onto the data bus and the way the ARM reads it.

pccard_regw

The index is loaded with the write register offset and the data is immediately written to the specified register. Unlike reads, there is no disparity in write operations between ARM and Vadem.

Open Access

pccard_leds

The Vadem chip set's unique registers are all locked after any reset. Therefore, the first action when using the chip is to unlock the registers by writing two initialization bytes to the base index register.

The General Purpose IO (GPIO) Configuration registers and the Miscellaneous Registers are all unique registers and must be accessed to control the GPIO bits.

When the registers are unlocked, the GPIO Configuration Register is written to, in order to set up both card sockets A and B as outputs.

A finite length loop (count of 100) is executed. On each iteration, it sets the bit in either A or B's Miscellaneous registers according to the state of a toggle flag. The A channel bit is ANDed with the flag while the B channel bit is ANDed with its inverse. The flag is then inverted for the next loop iteration.

A delay is set up to allow the switching LEDs to be seen.

When the loop is terminated, the registers are locked to tidy up and to set the registers into the known reset state. This should only be done if no further access to the PC Card is required.

main

This is the entry point for the program and is required by the C library. It calls the demonstration module and then terminates the program.

Open Access

7.6 PC Card (PCMCIA) SRAM Memory Card Read/Write Example

The ARM Development Board has an on-board PC Card (PCMCIA) interface with two available slots. This example shows how to set up the interface to write to, and read from, an SRAM Memory card.

Note *This module will not function unless a PC SRAM card is available.*

7.6.1 pcmcsram.c

Include files

`stdio.h`, `stdlib.h`	The standard C libraries have been included to enable the example code to be compiled as a stand-alone program.
`nisa.h`, `adb.h`, `rps.h`	These includes define the required offset values for the peripherals and standard address definitions of the ARM Development Board.
`pcmcia.h`	The final include file contains some of the most useful bit definitions for the pcmcia interface. To use this file, refer to the Vadem VG-468 data manual for a complete explanation of system and register operations.

Local definitions

The defined local variables are specific to pcmcsram.c

The majority of the local defines are address values that are particular to the operation of the PC memory card. They include the memory window start and stop offsets, the system memory offsets and the memory base address.

Also included in this are the defines for the Vadem to ARM address and data manipulation as well as operational masks such as those needed to clear any high order bits from a word transfer when a byte value is required.

pccard_regr

The required register offset is passed to this function and is loaded into the index register. This register is then read and the value shifted from the Vadem bus output position to the byte read position required by the ARM. This is necessary because of an inconsistency between the way the Vadem chip set passes read data onto the data bus and the way the ARM reads it. This is covered further in the notes section. Please read these notes before continuing.

pccard_regw

The index is loaded with the write register offset and the data is immediately written to the specified register. Unlike reads, there is no disparity in write operations between ARM and Vadem.

Open Access

pccard_sram_test

This module does not function correctly unless a PC SRAM Card is present. Therefore, initialization starts with a check that a card is present.

Next, the power supply and Auto-detect Power Switching (APS) are enabled. APS detects which slot the card is inserted in and automatically switches on power to that slot.

The Output Enable is then switched to release outputs from tri-state.

When the power is on, the code module checks for a card in the required slot. The card slot is hard-coded in the initialization part of this module.

A loop is set up to monitor the Card Detect bit for the required slot and holds at this point until a card is inserted. If the card is already present then the function continues without accessing the loop.

When a card is detected, it is reset by clearing the card reset bit (active low) while a delay counter is run down. The reset bit is also set for a similar delay time. The card is then checked for the Card Ready bit being set.

Next, the PC memory card is set up for use. Window1 is chosen for this example (an arbitrary choice of 0-4 is allowed).

The window Start and Stop addresses are set up with both the data width (8-bits in this case) and the number of wait states required (full).

The card memory offset is then set up prior to the window being enabled.

The system can now write data to the Card RAM, in this case the data is the offset to the address from 0 to 0xFF in increments of 1. This demonstrates the required address manipulation for operation between the ARM bus and the Vadem chipset. This is explained in more detail in the notes at the end of this section.

When the data has been written, it is sequentially read back and the map echoed to the debug screen. The data is also checked for integrity. This read operation also demonstrates the address manipulation and the data manipulation required in read operations between the ARM and Vadem systems.

The above write-read cycle is repeated using the inverse of the offset for the data, the data decrements from 0xFF to zero as the offset increases. This data is, again, echoed to the debug screen and is checked for integrity. Any failures cause a fault message to be displayed.

This function then terminates.

main

This is the entry point for the program and is required by the C library. It calls the Demonstration module and then terminates the program.

Open Access

7.6.2 Accessing the Vadem PCMCIA controller

Vadem-to-ARM read modification

There is a fundamental incompatibility between the behavior of the ARM and the Vadem PCMCIA controller chip. This incompatibility arises because of the way the ARM expects sub (32-bit) word data to be presented on the data bus, and the way the Intel-compatible Vadem chip presents it.

When the ARM loads a byte-sized quantity, it expects the byte to be presented on the byte lane corresponding to that byte. For example, for a byte at A[1:0] = “00”, the ARM expects the data to be on D[7:0], for a byte at A[1:0] = “01”, ARM expects the data to be on D[15:8] and so on.

When performing a byte read, the ARM grabs the appropriate byte lane, rotates this byte so that it sits in D[7:0] of the destination register, and zero extends the remaining bits of the register.

For halfword loads, the ARM expects the data to be on the appropriate half of the data bus, ie. on D[15:0] for A[1] = ‘0’ and on D[31:16] for A[1] = ‘1’. Again, the ARM rotates the halfword such that it resides in the bottom 16 bits of the destination register.

The Vadem PCMCIA controller is an Intel-compatible part and has a different mechanism for presenting data. The data path of the Vadem chip is only 16-bits wide and for byte reads, the data is always presented on D[7:0].

These two different mechanisms mean that the ARM can not perform byte reads from the Vadem chip directly. Reads with A[1:0] = “00” work. However, reads with A[1:0] = “01” do not because ARM extracts the wrong byte of data.

The A[1] on the ARM is not connected to the Vadem chip SA[1] for reasons discussed later. Instead, the ARM’s A[2] is connected to the Vadem chip’s SA[1].

If a word read is performed by the ARM, the Vadem chip treats this as a byte access (Vadem only supports 16 and 8-bit transfers). However for non word-aligned addresses, the ARM attempts to rotate the read word so that the byte corresponding to the address resides in D[7:0] of the destination register. This means that to read a byte from A[1:0] = “00”, the ARM must perform a word read and mask out bits D[31:8]. To read a word from A[1:0] = “01”, the ARM must shift the word left 8-bits before masking D[31:8]. Reads with the ARMs A[1] = ‘1’ should be avoided as they require a different fix.

The Vadem chip control register is an 8-bit register located at A[0] = ‘1’, so a fixed 24-bit shift and mask is required to read it. A general Algorithm to check for this is given as:

```
ARM_BYTE = ((ARM_ADDR & 0x1)? (RAW_DATA >> 24): (RAW_DATA & 0xFF))
```

ARM-to-Vadem transfers

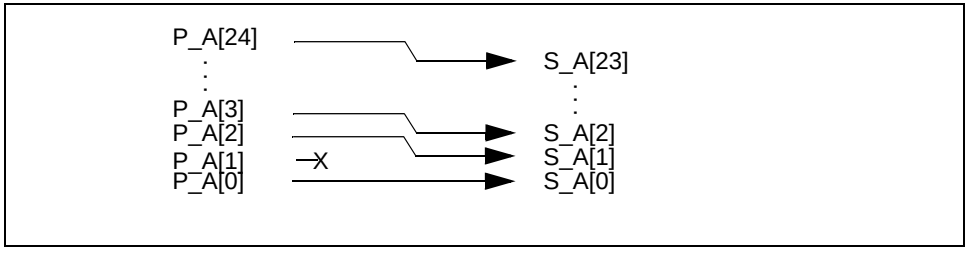
These transfers are not affected by the connection shown in ***Vadem-ARM address bus connection***, because the ARM outputs the data across all four byte lanes when writing to external systems. For byte transfers, the data is replicated across all four byte lanes. For halfword transfer, the data is output across the lower and upper halfwords.

Open Access

No modification to the ARM write code is required.

Vadem-ARM address bus connection

A[1] on the ARM is not connected to the Vadem's SA[1]. Instead, A[1] is unconnected and the remaining ARM address lines are shifted right one place as shown below:



Halfword reads by the ARM work correctly if A[1] = '0'. If A[1] is connected to the Vadem SA[1], reads from A[1] = '1' fail. If only word-aligned (A[1] = '0') accesses are performed, half of the data in the PCMCIA space is unreadable. For this reason, A[1] is not connected to the Vadem chip. Instead, A[2] is connected to SA[1], A[3] to SA[2] and so on. This allows halfword accesses to be performed from ARM word-aligned addresses.

This arrangement means that the addresses used by the ARM to access a particular PCMCIA location need modification. For a given card address, the required ARM address is given by:

```
ARM_ADDR = ((CARD_ADDR div 2) * 4) + (CARD_ADDR MOD 2)
```

or

```
ARM_ADDR = ((CARD_ADDR & ~0x1) << 1) + (CARD_ADDR & 0x1)
```

In summary, halfword accesses operate correctly only if the address is "fixed" as above.

The address increment for the next 16 bits of data is made four bytes rather than two.

Open Access

7.7 A ROM Resident Program

The code used to demonstrate ROM download and operation is a simple LED flashing program showing the use of three interrupt sources, two counter-timers and the parallel source interrupt.

7.7.1 Operation

The code is written to flash the four LEDs connected to the upper nibble of the parallel port in one of four sequences. The switches connected to the lower nibble of the parallel port control the speed of the sequence (switches 1 and 2) and the order of the sequence (3 and 4). The switches can be changed at any time, but are only read when the parallel interrupt is forced using the interrupt switch (the black button between the parallel and serial port connectors). The PC Card LEDs (A and B) also flash at the same speed as the sequence LEDs.

Sequence and speed are selected as shown in ***Table 7-1: Speed settings*** and ***Table 7-2: Sequence settings***.

S1	S2	Speed
ON	ON	Slow
OFF	ON	Medium slow
ON	OFF	Medium fast
OFF	OFF	Fast

Table 7-1: Speed settings

S3	S4	Sequence
ON	ON	Binary count
OFF	ON	Shifting left
ON	OFF	Alternating two LEDs
OFF	OFF	Shifting right

Table 7-2: Sequence settings

Open Access

7.7.2 Coding

The coding comprises three modules:

`stand.h` This file contains all the defines and the include files required in the C code. The include files are for the development board subsystems and RPS definitions as well as the nisa definitions and register operands for the PCMCIA and serial/parallel chip sets.

`boot.s` This is the system initialization source file required to ensure that the system boots correctly and that all of the stacks are initialized. This code should be used in all applications running from ROM.

The initial defines include the segments for the RAM stacks and some of the important system addresses (system RAM limit, ROM start address).

On entry to the boot code, the memory map is immediately flipped back and execution is carried out from the ROM space.

The Vector Block is then copied to the lower RAM to operate any further trapping required. In this case the only valid traps are Reset and IRQ. The rest of the vectors point to simple branch loops.

The separate stacks are then set up for IRQ, FIQ, ABT, UNDEF and finally SVC modes. Again, this is only done here to show a full setup since only the IRQ stack is used in this example.

Next, the data segments of the code including zero initialized data are copied to RAM. This data is also initialized.

Finally the system is set into USER mode, the required Interrupt bits (in this case IRQ only) are cleared, the stack pointer is set to the relevant stack and control is passed to the main C code entry point ‘C_Entry’

`stand_al.c` The first piece of code in this module is the IRQ Interrupt Handler. The label for this is imported into the boot source file and a branch is set up from the IRQ Interrupt vector.

The interrupt handler only operates on inputs from the Counter/Timers or the Parallel Source. If any interrupts are sensed, the source is checked and then the relevant flag is set to indicate the occurrence. The source interrupt is also cleared.

The main C routine initializes the system peripherals, counter/timers, parallel port, and the PC Card chip set.

The system is then set into an infinite loop which checks for instances of the interrupt flags being set.

When the loop is entered, an initialize flag is set to ensure that the switches are read on the first iteration. If the initialize flag is set or a parallel source interrupt is sensed, the switches (lower

Open Access

nibble of the parallel port) are read and the resulting value separated into speed (lower 2 bits) and sequence, which is shifted to give a value of 0 - 3.

A flag is set to initialize the sequence and the new speed is loaded into the counter/timers. The parallel source flag is reset with the initialize (first loop) flag which ensures that only the parallel source interrupt will activate this section of code.

The next check is for the timer1 interrupt flag. When this is sensed, the current state of the sequence is written to the highest nibble of the parallel port (LEDs). The changes are then applied to the sequence dependent upon which sequence is selected:

Binary	Increments the count between 1 and 15.
Shift Left	Shifts the single bit one place left unless bit 3 is set when no shift is carried out and the pattern starts at bit 0.
Alternate	Depending on the state of a toggle flag, either sets bits 0 and 2 and the toggle flag or, sets bit 1 and 3 and clears the toggle flag.
Shift Right	Shifts the single bit one place right unless bit 0 is set when no shift is carried out and the pattern starts at bit 3.

The final check in the loop is for a timer2 interrupt flag. When this is sensed, the two general purpose output bits of the PCMCIA chipset connected to PCA and PCB LEDs are toggled.

7.7.3 Command-line operation

Compiling the code

The code must be compiled specifically for ROM operation, because this requires some additional flags. To assemble the boot code, enter the following command line instruction:

```
armasm boot.s -li  -apcs 3/32bit/noswst -list boot.list -o boot.o
```

where:

`-li`	little-endian
`-apcs`	ARM procedure call standard
3	allow options
`/32bit`	use 32-bit ARM code
`/noswst`	do not check stacks

Remember that to compile the C code module, armcc must be used because this code contains an interrupt handler. If Thumb code is required, the handler must be put in a separate module and compiled in armcc - or tcc using the -32 option.

Open Access

The command-line instruction to compile the C Module is:

```
armcc -c stand_al.c  -li -apcs 3/32bit/noswst/nofp -i ..\include
```

where:

`-c`	compile only
`-li`	little-endian
`-apcs`	ARM procedure call standard
`3`	allow options
`/32bit`	use 32-bit ARM code
`/noswst`	do not check stacks
`/nofp`	do not use a frame pointer
`-i`	defines the include file directory
`..\include`	the files are stored in the \include directory

The above is written for a UNIX directory system. However, the ARM compiler accepts forward or back slash characters, and correctly specifies the path to the underlying operating system.

Both object files must then be linked using the armlink linker:

```
armlink  -BIN -BASE 0x4000000 -DATA 0x2000000 -Remove -NOZEROpad
    -o stand -First boot.o\(Boot\) stand_al.o boot.o
```

where:

`-BIN`	create a plain binary image
`-BASE`	read only code base location, ie. ROM base address
`0x4000000`	system address of ROM base on the development board
`-DATA`	data segment of Code - RAM address used in boot.s
`0x2000000`	base address of SRAM on the development board
`-Remove`	remove unused areas from the output [default]
`-NOZEROpad`	generate zero initialized segment in memory
`-o stand`	create output image labeled “stand”
`-First`	force the linker to place this section first ie. in lowest memory
`boot.o(Boot)`	place the boot area of boot.o at the lowest memory address

Note *In the above example, the UNIX command is given where escape characters ‘\’ are required for certain ASCII characters. The escape character is dependent upon your operating system.*

A binary image file is now created, and can be downloaded into the Flash EPROM on the development board.

Open Access

Downloading the code

You can now download the code as described in ***2.7.2 Flash downloading with armsd*** on page 2-32, using the filename `stand`.

When download is completed, you can check the code operation by setting the program counter to the base of the ROM area (0x4000000), and typing `go`.

7.7.4 Windows operation

Compiling the code

1. Select **New...→Project** from the File menu, and click on **OK**.
2. Enter the project name (for example, `stand`), and the project directory (for example, `ARM210\Examples\apm\stand`.
3. Select **Add Files to Project...** from the Project menu.
4. Select the assembler source `boot.s` and C source `stand_al.c` and click **Open**.
5. Select **Tool Configuration for stand.apj→cc→Set** from the Project menu.
6. Click on the **Procedure Call Standard** tab, clear the **Software Stack Check** and **Frame Pointer** boxes in the APCS3 Qualifiers section.
7. Click on the **Include Files** tab, enter `..\include` in the **Add directory** dialog box, and click on **Add**.
8. Click on **OK**.
9. Select **Tool Configuration for stand.apj→asm→Set** from the Project menu.
10. Click on the **Procedure Call Standard** tab, and clear the **Software Stack Check** in the APCS3 Qualifiers section.
11. Click on the **Include Files** tab, enter `..\include` in the **Add directory** dialog box, and click on **Add**.
12. Click on **OK**.
13. Select **Tool Configuration for stand.apj→armlink→Set** from the Project menu.
14. Click on the **Entry and Base** tab, enter `04x000000` in the **Read-Only** box and `0x2000000` in the **Read-Write** box.
15. Click on the **Image Layout** tab. In the Place at beginning of image section, enter `boot.o` in the **Object File** box, and `boot` in the **Area Name** box.
16. Click on **OK**.
17. Click on the **Build Debug** button at the bottom of the project window.

Downloading the code

You can now download the code as described in ***2.7.2 Flash downloading with armsd*** on page 2-32, using `stand` as the required ROM image.

Open Access

To use the code as boot code, the system must be set up for power-on remap using the REMAP link LK18, which is at the top of the main board, above and to the right of the frequency select switch S1. When this link is connected, the memory map changes at power-on to move the lowest part of ROM into the base of memory. This means that any code in ROM runs immediately at power-on.

Make the connection at LK18 and remove the EmbeddedICE JTAG connection (if used). This leaves the board in stand-alone mode. When the board is next powered up, the code runs as described above. To restart the code, press the RESET button (the red cap button between the parallel and serial port connectors).

Open Access

7.8 Flash Downloader

The ARM Development Board has capability for 8- or 16-bit ROM in the two chip carriers (32 pin DIP and 48 pin PLCC) U12 and U13 at the center of the board. These carriers can hold any form of ROM technology including Flash. This section describes an example of a RAM-resident Flash downloader that programs either the ATMEL AT29C040A (8-bit) or AT29C1024 (16-bit) Flash devices with an image held on the host computer.

Note *The Flash download utility supplied as part of the Software Development Toolkit 2.10 is an enhancement of the example code described in this section.*

7.8.1 Flash basics

Flash ROMs are programmable memory devices that contain programming and erasing circuitry for changing the memory while the chip is in the PCB.

Memory is programmed in set area segments. For the above two devices, this segment size is 256 bytes. Other devices have different sizes. These segments can only be programmed completely at one time, in the ATMEL devices. The downloaded binary image must be loaded 256 bytes at a time to the relevant address. The image usually starts at the base of the ROM memory and is loaded sequentially from there.

After each segment is downloaded, the Flash device programs the internal memory cells from the external latches where it stores the downloaded data.

In most cases, Flash devices have different modes of operation defined by a small command sequence. In the ATMEL devices, there are ROM operation, Product Identification and Data Protection modes. To change between any mode, a command sequence of two bytes (or words) is sent to separate addresses followed by a command byte (word) to initiate the change.

7.8.2 16-bit memory

This memory only operates over the halfword boundary, 16-bits of data per memory write. In this case, the lsb of the address bus is redundant as this defines the individual bytes. The ls address bit is, therefore, not connected to the 16-bit address lines. In the ATMEL devices, operation of the download is initiated by sending data to certain addresses within the mapped area of the ROM. When these addresses are passed to the 16-bit devices, they must be bit shifted to the left by one bit to ensure that they are on the word boundary required.

7.8.3 fl_wrt.c

Include files

`stdio.h`, `stdlib.h` The standard C libraries have been included to enable the example code to be compiled as a stand-alone program.

`mem.h` This include file contains the memory specific information for the ATMEL devices together with the ROM base address.

Open Access

Local definitions

The local defines are for byte and word offset values, the countdown timer value, the device programming sequence addresses and the command words for changing operation modes on both the byte and word devices.

7.8.4 Module explanations

main

main is the entry point to the program and the overall controller.

At initiation, the program must be called with the name of the image file to be downloaded. As with all C programs, this is passed from the command line to the main module as an argument (argv[1]). This can then be used for the file transfer.

First, the command line is checked for the inclusion of the image name. If there is none, the error is shown on the host and the program is terminated. Otherwise, the image file is checked for and opened. If there are any errors in this operation, an error message is displayed on the host and the program is terminated.

The fitted Flash device is then identified. If it is not recognized as either of the ATMEL devices, the program is terminated.

The input image file is echoed to the host screen and the data from the image is then downloaded to the Flash device defined in the identification code. As each 256-byte sector is programmed, the start address is incremented by 256 ready for the next sector. When the image data has been used up, any remaining words are filled with an alternating pattern to indicate the end of code.

When all of the image is downloaded, the image file path is closed and the download program is terminated.

write_sector

The arguments passed into this module are the start address to write to, the size of the data segment to be written, and the array of data to be written to the Flash segment.

This module begins by checking that the input sector size is not greater than the specified maximum. If it is, the program terminates.

The sector and the size of the sector are echoed to the host.

All data written to the Flash EPROM is written in protected mode to ensure that the code cannot be accidentally overwritten. When code protection is set on the ATMEL devices, it must be used on all sectors (see ***Table 7-3: ATMEL protection mode sequence*** on page 7-26). Otherwise, protection can be switched off for the whole device. The chip is set up in protected mode by writing the command sequence:

Open Access

Address	Byte Code	Word Code
0x5555	0xAA	0xAAAA
0x2AAA	0x55	0x5555

Table 7-3: ATMEL protection mode sequence

This routine is used for all changes of state in the ATMEL devices. The address given is the offset from the device 0x0000 address base. In the ARM Development Board, this is at location 0x4000000 in the memory map. The addresses given above need to be bit-shifted left by one bit for the 16-bit 1024 device and then added to the device base address.

To set Protected mode, the command byte (word) 0xA0 (0xA0A0) must be immediately sent to address Base + 0x5555. Again the added address must be bit-shifted. Following this sequence, the data can be downloaded to the device for the whole sector, a word at a time.

When all the sector data is downloaded, the device is left to program its memory cells. A check is performed on the final byte programmed to see when it has been set in memory before the program proceeds and the next sector is downloaded.

This process repeats until all of the segments are downloaded.

Identify

This module checks to see if the device to be programmed is recognized. If it is, the width of the data bus is checked.

The module initially reads the lowest addressed word for comparison with the identification register. When identification is switched on, this register becomes the lowest location word. If the switch to identification mode does not occur then the device will act as standard memory.

The device sequence is sent to the device and a 10ms timer is run down to give the device time to change state.

The module then carries out a byte device check. If this is successful then the word at Base + 0x0000 will contain the product information. The device can then be recognized as a byte device and the manufacturer and device code can be obtained.

If the base word is the same as that saved prior to sending the mode change sequence, the device is not a byte operation device and so a word sequence is initiated.

The word device addresses have to be set by left-shifting the offset addresses and then adding these to the base ROM addresses. In the word case, the mode change sequence data is 16-bit data. Again a 10ms timer is run to allow time for change. In the 16-bit device mode, the manufacture and device identification information is read from the base address two half words, still as the least significant byte of each half word.

Open Access

The device and manufacturer information is then compared with known values and, if recognized, the sector size, maximum sector size and word flags are set for the device.

The product identification is then switched off to return the device to standard operation.

7.8.5 Command-line operation

Compiling the program

The Flash downloader program requires access to:

- the standard C libraries
- fl_wrt.c
- mem.h

To initiate compilation type:

```
armcc -ARM7T -o flash -i ..\include fl_wrt.c
```

where:

`-ARM7T`	ensures halfword support for the compiled code
`-o flash`	makes the program image “flash”
`-i ..\include`	adds the include file directory to the include search path

Running the program

Initiate the Flash ROM downloader by sending the program into RAM by typing:

```
armsd [options] flash imagefile
```

where *`imagefile`* is the required binary image, and `[options]` are defined by the debugging system you are using (see ***2.4.3 Configuring the armsd debugger to use EmbeddedICE*** on page 2-18, or the relevant sections in ***2.5 Communicating with Angel*** on page 2-20).

7.8.6 Windows operation

Create a new project as an ARM executable image and add the file `fl_wrt.c` to the source directory. Configure the compiler for ARM7T architecture 4T.

This will now compile and link the executable for the target.

Open Access

Example Code Tutorials

Open Access

8 Benchmarking on the ARM Development Board

8.1	Dhrystone Benchmarking Test Function	8-2
8.2	Full Results	8-10

Open Access

Benchmarking on the ARM Development Board

8.1 Dhrystone Benchmarking Test Function

This program set demonstrates the use of Dhrystone standard tests on an ARM processor. The operation of this benchmark test is documented in the *ARM Software Development Toolkit User Guide* (ARM DUI 0040).

There are several reasons for porting this code to the ARM Development Board:

- Dhrystone can be run on different processors to give an overall rating for each against a stable benchmark. This provides comparisons between ARM and other processors.
- When run on the development board, the Dhrystone benchmark can give performance ratings for different peripheral configurations and clock speeds. Performance-critical routines can be identified and the system configuration optimized for the most efficient use of memory.
- The porting of this code to the development board includes the use of an internal source for the Dhrystone clock. Using external clock sources, Dhrystone tests may be inaccurate when different numbers of loop iterations are used. The development board counter timer produces an accurate clock source independent of any external influences.

This chapter explains the differences between the standard Dhrystone program set and this superset which allows the test to be run on the ARM Development Board. It also discusses results obtained from the tests and the relevance of these results when designing a system.

8.1.1 Differences

There are several differences between the standard suite of test modules (`dhry_1.c`, `dhry_2.c`, and `dhry.h`), and the ARM Development Board versions (`dhry_1pt.c`, `dhry_2pt.c`, `dhrypt.h`, and the timer defining files `timerpt.c` and `timerpt.h`). These files have the following changes:

`dhry_1pt.c` — This file has the addition of an extra clock source, defined if the external IRQ_CLOCK flag is set on compilation. This flag also adds an extra function call in main() to initialize the internal counter timer and uses calls to the irq_clock() to return values for the test begin and end timing.

`dhry_2pt.c` — This file has no changes from its original.

`dhrypt.h` — Uses the irq_CLOCK compiler flag to include the `timerpt.h` file in the module.

`timerpt.c` — This module must be compiled using the armcc compiler whether it is being used with Thumb or ARM code. This is discussed further in the compilation section.

The module contains the code required to set up and operate the development board counter timer as an accurate clock reference for the Dhrystone test.

The timer code includes an IRQ handler (TimerIncrement) which

Open Access

increments a counter every time a timer interrupt is detected. This handler is installed from the C code loading the interrupt vector with a LDR pc,[pc,#offs]. The offset is the location of the branch address of the interrupt function. There is no limit on the address of the interrupt handler as the vector branch address is held in local storage.

The development board timer is initialized by a routine in this module (init_timer) that disables all interrupts, initializes the timer by clearing any interrupt pending, and loads the timer load register with the value calculated from SYSTEM_CLOCK / 16 / 100. For a 20MHz clock, the timer will be loaded with 12500 or 0x30d4.

The timer is then set and enabled in periodic mode with a prescale value of 16. Finally the system interrupts are enabled for the timer.

Control is then passed exclusively to the Dhrystone code which queries the timer value by getting the counter value that is incremented in the interrupt handler. This gives an accurate clock value for the Dhrystone test.

`timerpt.h` This include file defines the clock tick value at 100.

8.1.2 Compiling the code using the command-line

The code can be compiled in two different states for four different clock speeds. You specify the clock speed on any compilation of the `timerpt.c` file by setting the clock speed flag as an external flag using the -D option, as shown in ***Table 8-1: Setting the Dhrystone system clock speed***.

System Clock (MHz)	Option to use
4	-DCLK_4
8	-DCLK_8
16	-DCLK_16
20	-DCLK_20

Table 8-1: Setting the Dhrystone system clock speed

If no -D flag is specified, the default speed is 20MHz.

The ARM Development Board system clock (MCLK) must also be set to the same speed as the compiler option. MCLK is set by the switch labeled S1 on the top left-hand side of the mother board, close to the daughter board. See ***Table 6-1: System Clock settings*** on page 6-4.

Open Access

Dhrystone can also be compiled for two different modes of operation:

- ARM-state operation (32 bit-code)
- Thumb-state operation (16-bit code)

The ARM Development Board can run in either or a combination of both.

However, `timerpt.c` must be compiled as ARM code since it includes a C code interrupt routine that cannot be handled by the Thumb C Compiler. This is because of complications when returning from an interrupt.

Thumb produces smaller code size while ARM produces higher performance code. The differences between Thumb and ARM can be seen by compiling the code for both states, and comparing code size and Dhrystone performance.

The following examples show these differences and assume a 20MHz clock speed:

Thumb code

The Thumb C compiler (tcc) cannot produce code to handle C code interrupt handlers. Therefore, all interrupt handler segments (in this case timerpt.c) must be compiled using the ARM C compiler. Since both ARM and Thumb code is required, interworking between the two states must be defined in the compiler directives. For example:

```
tcc -c -Otime -apcs /interwork -DIRQ_CLOCK dhry_1pt.c -o dhry_1pt.o
       -i ..\include
tcc -c -Otime -DIRQ_CLOCK dhry_2pt.c -o dhry_2pt.o -i ..\include
```

where:

`-c:`	compile only, no link.
`-Otime`	optimize for time
`-apcs /interwork`	call standard set for interworking
`-DIRQ_CLOCK`	define preprocess symbol IRQ_CLOCK during compilation
`-o dhry1_pt.o`	compile to object file `dhry_1pt.o`
`-i ..\include`	points to the include file directory

Note *When you compile these modules, some warnings are generated. This is because of ANSI C checking on modules written in K and RC. You can switch off the warnings using the -w flag.*

To compile the Interrupt Handler module, use armcc:

```
armcc -c -Otime -apcs /interwork/noswst/nofp -DCLK_20 timerpt.c -o
       timerpt.o -i ..\include
```

where:

`-apcs /noswst`	no software stack checking
`/nofp`	no frame pointer
`-DCLK_20`	define preprocess symbol for 20MHz clock

Open Access

Now link the files using the ARM Linker command:

```
armlink -o icedhry -info totals dhry_1pt.o dhry_2pt.o timerpt.o
      armlib_i.16l
```

where:

`-o icedhry`	defines the output image file
`-info totals`	show the total code and data sizes

The path for `armlib_i.16l` may be required if it is not in the system path.

ARM code

To compile the code for ARM operation, the suggested compile and link options are:

```
armcc -c -Otime -DIRQ_CLOCK dhry_1pt.c -o dhry_1pt.o -i ..\include
armcc -c -Otime -DIRQ_CLOCK dhry_2pt.c -o dhry_2pt.o -i ..\include
armcc -c -Otime  -DCLK_25 timerpt.c -o timerpt.o -i ..\include
armlink -o icedhry -info totals dhry_1pt.o dhry_2pt.o timerpt.o
armlib.32l
```

In this instance, there are no special interworking calls and a basic compilation is shown. The `-info` totals flag will show the overall code sizes for comparison with the Thumb code compiled above.

As before, you may have to insert the path for `armlib.32l` if it is not in the system path.

8.1.3 Using ARM Windows Project Manager to compile

You compile the code separately for ARM- and Thumb-state examples, as described in the following sections. If ARM Project Manager (.apj) files are included with the example suite, they can be used to generate the example code. The .apj files will be converted if compiled for a previous version of the APM.

ARM-state only code

Select the initial project type as the ARM executable image, and copy the files `dhry_1pt.c`, `dhry_2pt.c`, and `timerpt.c` into the sources directory.

1. Highlight the `dhry_1pt.c` file in the source directory.
2. Select the **cc** (compiler) option from the Tools menu.
3. Enter the flag `-DIRQCLOCK` in the **Command line** text box.
4. Repeat steps 1 to 3 for `dhry2pt.c`.
5. Repeat steps 1 to 2 for `timerpt.c`, and enter the flag `-DCLK_20` (for the 20MHz clock speed).

The code will now compile an ARM-state image that can be run on the target.

Open Access

Thumb-state (Interworking) code

For the Thumb-state code, set the initial project as a **THUMB-ARM interworking image**. Copy files `dhry_1pt.c` and `dhry_2pt.c` into the THUMB-C directory, and copy `timerpt.c` into the ARM-C directory.

1. Highlight the `dhry_1pt.c` file in the source directory.
2. Select the **cc** (compiler) option from the Tools menu.
3. Enter the flag `-DIRQCLOCK` in the **Command line** text box.
4. Repeat steps 1 to 3 for `dhry2pt.c`.
5. Repeat steps 1 to 2 for `timerpt.c`, and enter the flag `-DCLK_20` (for the 20MHz clock speed).

The code will now compile an ARM-THUMB interworking image that can be run on the target.

Tests

The above compilations were carried out at the four different clock frequencies. The resultant code was downloaded and run on the ARM Development Board for both ARM- and Thumb-state code.

When the tests were run on the development board, the memory configuration was changed to emulate different types of memory in order to view its effect on the execution of code in the system.

Results

There are three areas for comparison from this benchmarking test:

- code density
- code performance.
- memory configuration

These are discussed in turn in the following sections. The full set of results is shown in ***8.2 Full Results*** on page 8-10. All results were generated using the ARM SDT 2.01.

Code density

The following tables show the outputs from the linker when using the -info totals option:

Open Access

	Code size	Inline data	Inline strings	"Const" data	RW data	0-Init data	Debug data
Object totals	1704	76	1540	0	48	10200	388
Library totals	26128	900	732	128	700	1174	3336
Grand totals	27832	976	2272	128	748	11374	3724

Table 8-2: Thumb code size results

	Code size	Inline data	Inline strings	"Const" data	RW data	0-Init data	Debug Data
Object Totals	2512	36	1536	0	52	10200	132
Library Totals	34400	400	736	128	700	1176	416
Grand Totals	36912	436	2272	128	752	11376	548

Table 8-3: ARM code size results

16-bit Thumb code is more dense than 32-bit ARM code. The ratio is not exactly 2:1 for a number of reasons:

- Thumb code, although optimized, is not as efficient as the ARM code. There are occasions where one ARM command can only be accomplished by two, or even three Thumb commands.
- In this example, there is also the interrupt handler and its associated code which has to be compiled as ARM code. This affects the overall code density.

Code performance

The following results were obtained by running the code generated above on the ARM Development Board. The final values shown below are for board operation at 20MHz.

State	Frequency (MHz)	Cycles	µseconds per run	Dhrystones per second
Thumb	20	35000	84.6	11824.3

Open Access

State	Frequency (MHz)	Cycles	µseconds per run	Dhrystones per second
ARM	20	35000	73.7	13565.9

Table 8-4: Performance comparison between ARM and Thumb states

The higher efficiency of ARM code is shown as an increased Dhrystones/sec value over Thumb code. This is because features such as conditional execution and data shift during one instruction are available in ARM but not in Thumb.

The conclusion from these comparisons is that Thumb code offers a reduction in code size at the price of a small performance loss. ARM code offers better performance, but at the price of extra code.

Further examples of the Dhrystone tests are shown, for different clock speeds in ***8.2 Full Results*** on page 8-10.

Memory emulation

These results were taken by running code in each state (ARM and Thumb) six times at 25MHz and changing the memory configuration on each run.

The results show that when using 32-bit memory for both states, the fastest code execution occurs in ARM state with a small operational overhead in Thumb state, as discussed above.

As memory data size is reduced to 16-bit wide, the Thumb code becomes more efficient than ARM code. Although the ARM code almost doubles execution time, Thumb code loses very little performance. This is mainly because of the processor instruction and data fetch bottleneck that narrow memory presents to the ARM.

32-bit accesses always take one fetch to retrieve instructions or data from 32-bit memory. When the memory data path is reduced to 16-bits, 32-bit ARM code instructions and any 32-bit data writes require two memory accesses. Thumb code still requires only one fetch cycle per instruction and so retains its performance. The slight discrepancy in Dhrystone result between 32- and 16-bit memory for Thumb code can be attributed to 32-bit data writes and the interrupt handler. Thumb code should, theoretically, operate at the same speed if there were no full word data accesses and if the C interrupt handler could be compiled to Thumb.

When 8-bit memory is used, both ARM and Thumb code efficiencies are reduced by a factor of two. Again, this is because the number of fetch and write operations required to carry out instructions is doubled.

Increasing the number of wait cycles for memory operation gives a constant reduction in execution speed with no difference between the memory sizes.

8.1.4 Conclusion

This exercise shows that benchmarking is a very important part of system design. It identifies the most efficient methods for coding and memory design in order to get the optimum performance out of a system.

Open Access

Memory is currently a major proportion of system cost. Running these benchmarks shows that to reduce code storage and cost, you could use Thumb code so that 16-bit memory can be used, with the possible addition of some local 32-bit memory for interrupt handling and 32-bit data storage.

Careful examination of code functions could even enable you to use 8-bit memory for code that is not time critical.

The flexibility of the ARM Development Board enables you to determine the optimum memory speed and width for your code experimentally, before you commit to building the target hardware.

Open Access

8.2 Full Results

8.2.1 Code size comparison

	Code size	Inline data	Inline strings	"Const" data	RW data	0-Init data	Debug data
Object totals	2512	36	1536	0	52	10200	132
Library totals	34400	400	736	128	700	1176	416
Grand totals	36912	436	2272	128	752	11376	548

Table 8-5: Dhrystone ARM code size

	Code size	Inline data	Inline strings	"Const" data	RW data	0-Init data	Debug data
Object totals	1704	76	1540	0	48	10200	388
Library totals	26128	900	732	128	700	1174	3336
Grand totals	27832	976	2272	128	748	11374	3724

Table 8-6: Dhrystone Thumb code size

8.2.2 Code performance comparison

Clock (MHz)	Number of cycles	µseconds per run	Dhrystone per second
4	35000	368.6	2713.2
8	35000	184.0	5434.8
16	35000	91.7	10903.4
20	35000	73.7	13565.9

Table 8-7: Dhrystone performance in ARM state

Open Access

Clock (MHz)	Number of cycles	µseconds per run	Dhrystone per second
4	35000	422.2	2364.9
8	35000	211.1	4736.1
16	35000	105.7	9459.5
20	35000	84.6	11824.3

Table 8-8: Dhrystone performance in Thumb state

8.2.3 Varying memory width and speed

Memory width	Memory speed	µseconds per Dhrystone	Dhrystone per second
32-bit	2 cycle	73.7	13565.9
16-bit	2 cycle	133.1	7510.7
8-bit	2 cycle	252.0	3968.3
32-bit	5 cycle	165.1	6055.4
16-bit	5 cycle	314.0	3184.7
8-bit	5 cycle	612.0	1634.0

Table 8-9: Dhrystone performance at 20MHz in ARM state for various memories over 35000 iterations

Open Access

Memory width	Memory speed	µseconds per Dhrystone	Dhrystone per second
32-bit	2 cycle	84.6	11824.3
16-bit	2 cycle	100.0	10000.0
8-bit	2 cycle	183.4	5451.7
32-bit	5 cycle	189.1	5287.0
16-bit	5 cycle	227.7	4391.5
8-bit	5 cycle	437.4	2286.1

Table 8-10: Dhrystone performance at 20MHz in Thumb state for various memories over 35000 iterations

Open Access

9 ARM Development Board Diagnostic Utility

9.1	Introduction	9-2
9.2	selftest	9-3
9.3	diagnose	9-4

Open Access

9.1 Introduction

The ARM Development Board diagnostic utilities, selftest and diagnose, are provided in binary image form only. The utilities are included to enable you to test the board operation to different levels and to give confidence in the integrity of the board. Source files and code explanations are not provided.

Open Access

9.2 selftest

Note *The ATMEL AT29C040A Flash device provided with this board has an access time of 120ns. This means that at 20MHz, the clock timing is 50ns. Therefore, you must set up a minimum of three cycles for the system to operate correctly. As explained in* ***6.5 Setting the EPROM/Flash Speed and Width*** on page 6-7*, the access time is defined by the EPROM SELECT link field (LK 6) positions 1-2 and 3-4. Please refer to this section to set the Flash access time correctly.*

The selftest utility tests all the on-board peripheral registers and ensure that there is communication available between them and the processor. The code tests the Remap Controller, the Interrupt Controller, Counter Timers, Serial Comms System, PCMCIA Controller and Flash Memory.

There is no operator control of this code. Progress is indicated by the PCA and PCB LEDs flashing to indicate test results. Both LEDs flashing together indicates a test PASS. If the LEDs flash individually (PCA on, both LEDs off, PCB on, both LEDs off), there is a fault on the board and you should execute the diagnose utility to find out which subsystem is faulty.

If the selftest function is required and the code has been corrupted, you can download and operate the image from power-up or system reset by using the Flash downloader utility, see ***2.7 Using the Flash Downloader*** on page 2-31.

Open Access

9.3 diagnose

This utility is supplied as a binary file that you can download and run using a armsd or ADW.

The code carries out all the selftest tests, and some additional system tests to check the board operation more fully. A full results listing is displayed and saved at the end of the test. Operator intervention is required throughout the test.

Do not run this utility on a system with any other user code installed because some RAM and Flash locations may be overwritten during the tests.

To check the system fully from the command line, first ensure that the current directory name is `diag` and that both “devmem” and “diagnose” images are present. Results are written to `result.txt`. If there is an existing result.txt file from a previous run of the diagnostics that you require for future use, rename it or store it in a new directory.

Some of the tests require that the development board is set up for testing to enable external subsystems to be checked or observed in operation. The required setup is:

1. Ensure that all links in LK11 (behind the parallel port connector, bottom right-hand side of board) are made. This requires eight link connectors, each to be placed across a pair of contacts. This connects the switches S3 and LEDs D7-D10 to the parallel port data lines.
2. Set up a serial loop cable between the two serial ports, SERIAL A and SERIAL B. This link is a simple three-wire cable connector with a twist between pins 2 and 3 and a straight connection to pin 5 between the two D-type connectors.

9.3.1 Using the command line

Initialize the armsd debugger for use with EmbeddedICE (see ***2.4.3 Configuring the armsd debugger to use EmbeddedICE*** on page 2-18).

When the `armsd:` prompt is displayed type:

```
$vector_catch=0
get devmem 0
pc=0
go
```

This runs the memory check for the SSRAM, SRAM and DRAM. Messages for the test results are echoed to the screen. If there is no DRAM connected then the DRAM test fails and reports the failure to the screen. The following results are taken from a board as shipped, with no DRAM installed:

```
ARM7TDMI Development Card Memory Test
Checking SRAM...OK
Checking SSRAM memory...OK
Checking DRAM...Failed
Program terminated normally at PC = 0x00000008
0x00000008: 0xea00009c  .... :    b        0x280
```


Open Access

When the memory tests are complete, type the following at the `armsd:` prompt to download and run the system diagnostic code:

```
load diagnose
go
```

The full diagnostics testing execute and the results are shown on the screen and saved in a file called `results.txt`.

9.3.2 Using the Windows debugger

The debugger must be configured to operate with EmbeddedICE (see ***2.4.2 Configuring the ARM Debugger for Windows to use EmbeddedICE*** on page 2-15).

When the debugger and EmbeddedICE are configured, the code Image must be sent to the ARM system from the host. If the debugger was started from the Project Manager, the code must be reloaded. If the debugger was initiated alone, select the **Load Image** option from the File menu, and load the “diagnose” image.

To execute the code, click the **Go** button on the toolbar, or select **Go** from the Execute menu.

9.3.3 Program operation

Some tests require you to observe the board (LEDs) or ensure that some setup is made (such as insertion of the PC SRAM card). When you are required to interact with the process, the program operation is halted by a loop and the screen instructs you to press the interrupt button (the black-capped button between the serial and parallel ports). The red-capped button is the card reset. When you press the interrupt button, an interrupt is generated and the program continues.

You can bypass the PC SRAM card checks if these cards are not available, by pressing the interrupt button.

When the diagnostic program finishes, both PCA and PCB LEDs flash to indicate the results of the test.

- Both LEDs flashing together indicates a test PASS.
- LEDs flashing individually (PCA on, both LEDs off, PCB on, both LEDs off), indicates a fault on the board. The subsystem level cause is indicated on the screen.

You can view all result outputs in the text file `results.txt`, in the working directory.

Open Access

Open Access

A ST16C552 UART with FIFO and Parallel Printer Port

Note *The content of the ST16C552 programmer's data sheet has been reproduced with the express permission of EXAR Corporation.*

(c) 1995,1996 EXAR Corporation

A.1	Description	A-2
A.2	Serial Port Programming	A-3
A.3	Printer Port Programming	A-15

Open Access

ST16C552 UART with FIFO and Parallel Printer Port

A.1 Description

The ST16C552 is a dual universal asynchronous receiver and transmitter with 16 byte transmit and receive FIFO and a bi-directional CENTRONICS type parallel printer port. A programmable baud rate generator is provided to select transmit and receive clock rates from 50 bps to 1.5 Mbps.

The ST16C552 on board status registers provides the error conditions, type and status of the transfer operation being performed. Included is complete MODEM control capability, and a processor interrupt system that may be software tailored to the user's requirements. The ST16C552 provides internal loop-back capability for on board diagnostic testing.

Open Access

A.2 Serial Port Programming

A2	A1	A0	READ MODE	WRITE MODE
0	0	0	Receive Holding Register	Transmit Holding Register
0	0	1		Interrupt Enable Register
0	1	0	Interrupt Status Register	FIFO Control Register
0	1	1		Line Control Register
1	0	0		Modem Control Register
1	0	1	Line Status Register	
1	1	0	Modem Status Register	
1	1	1	Scratchpad Register	Scratchpad Register
0	0	0		LSB of Divisor Latch
0	0	1		MSB of Divisor Latch

Table 9-1: Programming table for serial ports

Open Access

ST16C552 Accessible Registers

A2	A1	A0	Reg-ister	BIT-7	BIT-6	BIT-5	BIT-4	BIT-3	BIT-2	BIT-1	BIT-0
0	0	0	RHR	bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
0	0	0	THR	bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
0	0	1	IER	0	0	0/ special mode	0	modem status interrupt	receive line status interrupt	transmit holding register	receive holding register
0	1	0	FCR	RCVR trigger h(MSB)	RCVR trigger (LSB)	0	0	DMA mode select	XMIT FIFO reset	RCVR FIFO reset	FIFO enable
0	1	0	ISR	0/ FIFOs enabled	0/ FIFOs enabled	0	0	int priority bit-2	int priority bit-1	int priority bit-0	int status
0	1	1	LCR	divisor latch enable	set break	set parity	even parity	parity enable	stop bits	word length bit-1	word length bit-0
1	0	0	MCR	0/power down	0	0	loop back	INT enable	Not used	RTS*	DTR*
1	0	1	LSR	0/ FIFO error	trans. empty	trans. holding empty	break interrupt	framing error	parity error	overrun error	receive data ready
1	1	0	MSR	CD	RI	DSR	CTS	delta CD*	delta RI*	delta DSR*	delta CTS*
1	1	1	SPR	bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
0	0	0	DLL	bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
0	0	1	DLM	bit-15	bit-14	bit-13	bit-12	bit-11	bit-10	bit-9	bit-8

Table A-1: Accessible registers

DLL and DLM are accessible only when LCR bit-7 is set to "1"

Open Access

A.2.1 Register Functional Descriptions

Transmit and receive holding register

The serial transmitter section consists of a Transmit Hold Register (THR) and Transmit Shift Register (TSR). The status of the transmit hold register is provided in the Line Status Register (LSR). Writing to this register (THR) will transfer the contents of data bus (D7-D0) to the Transmit holding register whenever the transmitter holding register or transmitter shift register is empty. The transmit holding register empty flag will be set to “1” when the transmitter is empty or data is transferred to the transmit shift register. Note that a write operation should be performed when the transmit holding register empty flag is set.

On the falling edge of the start bit, the receiver internal counter will start to count 7 1/2 clocks (16x clock) which is the center of the start bit. The start bit is valid if the RX is still low at the mid-bit sample of the start bit. Verifying the start bit prevents the receiver from assembling a false data character due to a low going noise spike on the RX input. Receiver status codes will be posted in the Line Status Register.

FIFO interrupt mode operation

When the receive FIFO (FCR BIT-0=1) and receive interrupts (IER BIT-0=1) are enabled, receiver interrupt will occur as follows:

A) The receive data available interrupts will be issued to the CPU when the FIFO has reached its programmed trigger level; it will be cleared as soon as the FIFO drops below its programmed trigger level.

B) The ISR receive data available indication also occurs when the FIFO trigger level is reached, and like the interrupt it is cleared when the FIFO drops below the trigger level.

C) The data ready bit (LSR BIT-0) is set as soon as a character is transferred from the shift register to the receiver FIFO. It is reset when the FIFO is empty.

FIFO polled mode operation

When FCR BIT-0=1; resetting IER BIT 3-0 to zero puts the ST16C552 in the FIFO polled mode of operation. Since the receiver and transmitter are controlled separately either one or both can be in the polled mode operation by utilizing the Line Status Register.

A) LSR BIT-0 will be set as long as there is one byte in the receive FIFO.

B) LSR BIT4-1 will specify which error(s) has occurred.

C) LSR BIT-5 will indicate when the transmit FIFO is empty.

D) LSR BIT-6 will indicate when both transmit FIFO and transmit shift register are empty.

E) LSR BIT-7 will indicate when there are any errors in the receive FIFO.

The ST16C552 requires to have two step FIFO enable operation in order to enable receive trigger levels.

Open Access

Programmable baud rate generator

The ST16C552 contains a programmable Baud Rate Generator that is capable of taking any clock input from DC-24 MHz and dividing it by any divisor from 1 to 2^{16} -1. The output frequency of the Baudout* is equal to 16X of transmission baud rate (Baudout*=16 x Baud Rate). Customize Baud Rates can be achieved by selecting proper divisor values for MSB and LSB of baud rate generator.

Interrupt Enable Register (IER)

The Interrupt Enable Register (IER) masks the incoming interrupts from receiver ready, transmitter empty, line status and modem status registers to the INT output pin.

IER BIT-0 0=disable the receiver ready interrupt.
1=enable the receiver ready interrupt.

IER BIT-1 0=disable the transmitter empty interrupt.
1=enable the transmitter empty interrupt.

IER BIT-2 0=disable the receiver line status interrupt.
1=enable the receiver line status interrupt.

IER BIT-3 0=disable the modem status register interrupt.
1=enable the modem status register interrupt.

IER BIT -4 Not used. This bit is set to logic zero.

IER BIT -5 0=Normal.
1=Enable special mode (Power down).

IER BIT 6-7 All these bits are set to logic zero.

Interrupt Status Register (ISR)

The ST16C552 provides four level prioritized interrupt conditions to minimize software overhead during data character transfers. The Interrupt Status Register (ISR) provides the source of the interrupt in prioritized matter. During the read cycle the ST16C552 provides the highest interrupt level to be serviced by CPU. No other interrupts are acknowledged until the particular interrupt is serviced. The following are the prioritized interrupt levels:

Open Access

P	D3	D2	D1	D0	Source of the interrupt
1	0	1	1	0	LSR (Receiver Line Status Register)
2	0	1	0	0	RXRDY (Received Data Ready)
2*	1	1	0	0	RXRDY (Receive Data time out)
3	0	0	1	0	TXRDY (Transmitter Holding Register Empty)
4	0	0	0	0	MSR (Modem Status Register)

Table A-2: Priority level

***RECEIVE TIME-OUT:**

This mode is enabled when STARTECH UART is operating in FIFO mode. Receive time out will not occur if the receive FIFO is empty. The time out counter will be reset at the center of each stop bit received or each time receive holding register is read. The actual time out value is T (**T**ime out length in bits)= 4 X P (**P**rogrammed word length) + 12. To convert time out value to a character value, user has to divide this number to its complete word length + parity (if used) + number of stop bits and start bit.

Example -A: If user programs the word length = 7, and no parity and one stop bit, Time out will be:

T = 4 X 7(programmed word length) +12 = 40 bits

Character time = 40 / 9 [(programmed word length = 7) + (stop bit = 1) + (start bit = 1)] = 4.4 characters.

Example -B: If user programs the word length = 7, with parity and one stop bit, the time out will be:

T = 4 X 7(programmed word length) + 12 = 40 bits

Character time = 40 / 10 [(programmed word length = 7) + (parity = 1) + (stop bit = 1) + (start bit = 1) = 4 characters.

ISR BIT-0 0=an interrupt is pending and the ISR contents may be used as a pointer to the appropriate interrupt service routine.

1=no interrupt pending.

ISR BIT 1-3 Logical combination of these bits, provides the highest priority interrupt pending.

ISR BIT 4-7 These bits are not used and are set to zero in ST16C450 mode. **BIT 6-7:** are set to “1” in ST16C552 mode.

Open Access

FIFO Control Register (FCR)

This register is used to enable the FIFOs, clear the FIFOs, set the receiver FIFO trigger level, and select the type of DMA signaling.

FCR BIT-0 0=Disable the transmit and receive FIFO.

1=Enable the transmit and receive FIFO.

This bit should be enabled before setting the FIFO trigger levels.

FCR BIT-1 0=No change.

1=Clears the contents of the receive FIFO and resets its counter logic to 0 (the receive shift register is not cleared or altered). This bit will return to zero after clearing the FIFOs.

FCR BIT-2 0=No change.

1=Clears the contents of the transmit FIFO and resets its counter logic to 0 (the transmit shift register is not cleared or altered). This bit will return to zero after clearing the FIFOs.

FCR BIT-3 0=No change.

1=Changes RXRDY and TXRDY pins from mode “0” to mode “1”.

Transmit operation in mode "0"

When ST16C552 is in ST16C450 mode (FCR bit-0=0) or in the FIFO mode (FCR bit-0=1, FCR bit-3=0) when there are no characters in the transmit FIFO or transmit holding register, the TXRDY* pin will go low. Once active the TXRDY* pin will go high (inactive) after the first character is loaded into the transmit holding register.

Receive operation in mode "0"

When ST16C552 is in ST16C450 mode (FCR bit-0=0) or in the FIFO mode (FCR bit-0=1, FCR bit-3=0) and there is at least 1 character in the receive FIFO, the RXRDY* pin will go low. Once active the RXRDY* pin will go high (inactive) when there are no more characters in the receiver.

Transmit operation in mode "1"

When ST16C552 is in ST16C550 mode (FCR bit-0=1, FCR bit-3=1) the TXRDY* pin will become high (inactive) when the transmit FIFO is completely full. It will be low if one or more FIFO locations are empty.

Receive operation in mode "1"

When ST16C552 is in ST16C550 mode (FCR bit-0=1, FCR bit-3=1) and the trigger level or the timeout has been reached, the RXRDY* pin will go low. Once it is activated it will go high (inactive) when there are no more characters in the FIFO.

FCR BIT 4-5 Not used.

Open Access

FCR BIT 6-7 These bits are used to set the trigger level for the receiver FIFO interrupt.

BIT-7	BIT-6	FIFO trigger level
0	0	01
0	1	04
1	0	08
1	1	14

Line Control Register (LCR)

The Line Control Register is used to specify the asynchronous data communication format. The number of the word length, stop bits, and parity can be selected by writing appropriate bits in this register.

LCR BIT1-0 These two bits specify the word length to be transmitted or received.

BIT-1	BIT-0	Word length
0	0	5
0	1	6
1	0	7
1	1	8

LCR BIT-2 The number of stop bits can be specified by this bit.

BIT-2	Word length	Stop bit(s)
0	5,6,7,8	1
1	5	1-1/2
1	6,7,8	2

LCR BIT-3 Parity or no parity can be selected via this bit.

0=no parity

1=a parity bit is generated during the transmission, receiver also checks for received parity.

LCR BIT-4 If the parity bit is enabled, LCR BIT-4 selects the even or odd parity

Open Access

format.

0=ODD parity is generated by forcing an odd number of 1's in the transmitted data, receiver also checks for same format.

1= EVEN parity bit is generated by forcing an even the number of 1's in the transmitted data, receiver also checks for same format.

LCR BIT-5 If the parity bit is enabled, LCR BIT-5 selects the forced parity format.

LCR BIT-5=1 and LCR BIT-4=0, parity bit is forced to "1" in the transmitted and received data.

LCR BIT-5=1 and LCR BIT-4=1, parity bit is forced to "0" in the transmitted and received data.

LCR BIT-6 Break control bit. It causes a break condition to be transmitted (the TX is forced to low state).

0=normal operating condition.

1=forces the transmitter output (TX) to go low to alert the communication terminal.

LCR BIT-7 The internal baud rate counter latch enable (DLEN).

0=normal operation.

1=select divisor latch register.

Modem Control Register (MCR)

This register controls the interface with the MODEM or a peripheral device (RS232).

MCR BIT-0 0=force DTR* output to high.

1=force DTR* output to low.

MCR BIT-1 0=force RTS* output to high.

1=force RTS* output to low.

MCR BIT-2 Not used.

MCR BIT-3 0=set INT output pin to three state mode.

1=set INT output pin to normal / active operating mode.

MCR BIT-4 0=normal operating mode.

1=enable local loop-back mode (diagnostics). The transmitter output (TX) is set high (Mark condition), the receiver input (RX) , CTS*, DSR*, CD*, and RI* are disabled. Internally the transmitter output is connected to the receiver input and DTR*, RTS*, MCR bit-2 and INT enable are connected to modem control inputs.

In this mode , the receiver and transmitter interrupts are fully operational. The Modem Control Interrupts are also operational, but the interrupts sources are now the lower four bits of the Modem Control Register instead of the four Modem Control inputs. The interrupts are still controlled by the IER .

MCR BIT 5-6 Not used. Are set to zero permanently.

Open Access

MCR BIT -7 0=Normal.

1=Power down mode. The IER Bit-5 has to be set before power down takes place.

Line Status Register (LSR)

This register provides the status of data transfer to CPU.

LSR BIT-0 0=no data in receive holding register or FIFO.

1=data has been received and saved in the receive holding register or FIFO.

LSR BIT-1 0=no overrun error (normal).

1=overrun error, next character arrived before receive holding register was emptied or if FIFOs are enabled, an overrun error will occur only after the FIFO is full and the next character has been completely received in the shift register. Note that character in the shift register is overwritten, but it is not transferred to the FIFO.

LSR BIT-2 0=no parity error (normal).

1=parity error, received data does not have correct parity information. In the FIFO mode this error is associated with the character at the top of the FIFO.

LSR BIT-3 0=no framing error (normal).

1=framing error received, received data did not have a valid stop bit. In the FIFO mode this error is associated with the character at the top of the FIFO.

LSR BIT-4 0=no break condition (normal).

1=receiver received a break signal (RX was low for one character time frame). In FIFO mode, only one zero character is loaded into the FIFO.

LSR BIT-5 0=transmit holding register is full. ST16C552 will not accept any data for transmission.

1=transmit holding register (or FIFO) is empty. CPU can load the next character.

LSR BIT-6 0=transmitter holding and shift registers are full.

1=transmitter holding and shift registers are empty. In FIFO mode this bit is set to one whenever the transmitter FIFO and transmit shift register are empty.

LSR BIT-7 0=Normal.

1=At least one parity error, framing error or break indication in the FIFO. This bit is cleared when LSR is read.

Open Access

Modem Status Register (MSR)

This register provides the current state of the control lines from the modem or peripheral to the CPU. Four bits of this register are used to indicate the changed information. These bits are set to “1” whenever a control input from the MODEM changes state. They are set to “0” whenever the CPU reads this register.

MSR BIT-0 Indicates that the CTS* input to the ST16C552 has changed state since the last time it was read.

MSR BIT-1 Indicates that the DSR* input to the ST16C552 has changed state since the last time it was read.

MSR BIT-2 Indicates that the RI* input to the ST16C552 has changed from a low to a high state.

MSR BIT-3 Indicates that the CD* input to the ST16C552 has changed state since the last time it was read.

MSR BIT-4 This bit is equivalent to RTS in the MCR during local loop-back mode. It is the compliment of the CTS* input.

MSR BIT-5 This bit is equivalent to DTR in the MCR during local loop-back mode. It is the compliment of the DSR* input.

MSR BIT-6 This bit is equivalent to MCR bit-2 during local loop-back mode. It is the compliment of the RI* input.

MSR BIT-7 This bit is equivalent to INT enable in the MCR during local loop-back mode. It is the compliment to the CD* input.

Note *Whenever MSR BIT3-0: is set to logic “1”, a MODEM Status Interrupt is generated.*

Scratchpad Register (SR)

ST16C552 provides a temporary data register to store 8 bits of information for variable use.

Open Access

ST16C552 External Reset Condition

REGISTERS	RESET STATE
IER	BITS 0-7=0
ISR	ISR BIT-0=1, ISR BITS 1-7=0
LCR	LCR BITS 0-7=0
MCR	MCR BITS 0-7=0
LSR	LSR BITS 0-4=0,
	LSR BITS 5-6=1 LSR, BIT 7=0
MSR	MSR BITS 0-3=0,
	MSR BITS 4-7=input signals
FCR	FCR BITS 0-7=0

Table A-3: Register reset state

SIGNALS	RESET STATE
TX A/B	High
RTS* A/B	High
DTR* A/B	High
INT A/B, P	Three state mode
RXRDY* A/B	High
TXRDY* A/B	Low

Table A-4: Signal reset state

Open Access

Baud rate generator programming

BAUD RATE	16 x CLOCK DIVISOR	% ERROR
50	2304	
110	1047	0.026
150	768	
300	384	
600	192	
1200	96	
2400	48	
4800	24	
7200	16	
9600	12	
19.2K	6	
38.4K	3	
56K	2	2.77
115.2K	1	

Table A-5: Baud rate generator programming table (1.8432 MHz clock)

Open Access

A.3 Printer Port Programming

A1	A0	IOW*	IOR*
0	0	PORT REGISTER	PORT REGISTER
0	1	I/O SELECT REGISTER	STATUS REGISTER *
1	0	CONTROL REGISTER	COMMAND REGISTER

Table A-6: Printer port programming table

* Reading the status register will reset the INTP output.

A.3.1 Printer Port Register Descriptions

PR BIT 7-0 PD7-PD0 bi-directional I/O ports.

Status Register

This register provides the state of the printer outputs and the interrupt condition.

SR BIT 1-0 Not used. Are set to “1” permanently.

SR BIT-2 Interrupt condition.
0= an interrupt is pending
This bit will be set to “0” at the falling edge of the ACK* input.
1= no interrupt is pending
Reading the STATUS REGISTER will set this bit to “1”.

SR BIT-3 ERROR* input state.
0= ERROR* input is in low state
1= ERROR* input is in high state

SR BIT-4 SLCT input state.
0= SLCT input is in low state
1= SLCT input is in high state

SR BIT-5 PE input state.
0= PE input is in low state
1= PE input is in high state

SR BIT-6 ACK* input state.
0= ACK* input is in low state
1= ACK* input is in high state

Open Access

SR BIT-7 BUSY input state.
0= BUSY input is in high state
1= BUSY input is in low state

Command Register

The state of the STROBE*, AUTOFDXT*, INIT, SLCTIN* pins, and interrupt enable bit can be read by this register regardless of the I/O direction.

COM BIT-0 STROBE* input pin.
0= STROBE* pin is in high state
1= STROBE* pin is in low state

COM BIT-1 AUTOFDXT* input pin.
0= AUTOFDXT* pin is in high state
1= AUTOFDXT* pin is in low state

COM BIT-2 INIT input pin.
0= INIT pin is in low state
1= INIT pin is in high state

COM BIT-3 SLCTIN* input pin.
0= SLCTIN* pin is in high state
1= SLCTIN* pin is in low state

COM BIT-4 Interrupt mask.
0= Interrupt (INTP output) is disabled
1= Interrupt (INTP output) is enabled

COM BIT 7-5 Not used. Are set to “1” permanently.

Control Register

Writing to this register will set the state of the STROBE*, AUTOFDXT*, INIT, SLCTIN pins, and interrupt mask register.

CON BIT-0 STROBE* output control bit.
0= STROBE* output is set to high state
1= STROBE* output is set to low state

CON BIT-1 AUTOFDXT* output control bit.
0= AUTOFDXT* output is set to high state
1= AUTOFDXT* output is set to low state

CON BIT-2 INIT output control bit.
0= INIT output is set to low state
1= INIT output is set to high state

Open Access

CON BIT-3 SLCTIN* output control bit.

0= SLCTIN* output is set to high state

1= SLCTIN* output is set to low state

CON BIT-4 Interrupt output control bit.

0= INTP output is disabled

1= INTP output is enabled

CON BIT-5 I/O select. Direction of the PD7-PD0 can be selected by setting or clearing this bit.

0= PD7-PD0 are set for output mode

1= PD7-PD0 are set for input mode

CON BIT 7-6 Not used.

CONTROL REGISTER (D5)	BIDEN	I/O SELECT REGISTER	PORT DIRECTION
X	0	AA Hex	Input mode
X	0	55 Hex	Output mode
0	1	X	Output mode
1	1	X	Input mode

I/O Select Register

Software controlled I/O select.

Bi-directional mode can be selected by keeping the BIDEN input in high state and setting CON BIT-5 to “zero or one”

Hardware/software I/O select.

Bi-directional mode can be selected by keeping the BIDEN input in low state and setting I/O SELECT register to “AA” Hex for input or “55” Hex for output.

Open Access

External reset condition

SIGNALS	RESET STATE
PD0-PD7	Low, output mode
STROBE*	High, output mode
AUTOFDXT*	High, output mode
INIT	Low, output mode
SLCTIN*	High, output mode

Table A-7: ST16C552 external reset condition

ST16C552 Printer Port Register Configurations

D7	D6	D5	D4	D3	D2	D1	D0
PD7	PD6	PD5	PD4	PD3	PD2	PD1	PD0

Table A-8: Port Register (read/write)

D7	D6	D5	D4	D3	D2	D1	D0
BUSY*	ACK	PE	SLCT	ERROR STATE	IRQ	1	1
					1= No interrupt 0= Interrupt		

Table A-9: Status Register (read-only)

Open Access

D7	D6	D5	D4	D3	D2	D1	D0
1	1	1	IRQ ENABLE	SLCTIN*	INIT	AUTO-FDXT*	STROBE*
			0= IRQ disabled 1= IRQ enabled				

Table A-10: Command Register (read-only)

D7	D6	D5	D4	D3	D2	D1	D0
X	X	I/O SELECT	IRQ MASK	SLCTIN*	INIT	AUTO-FDXT*	STROBE*
	0=Output1=Input		0=INTP output disabled 1=INTP output enabled				

Table A-11: Control Register (write-only)

Open Access

ST16C552 UART with FIFO and Parallel Printer Port

Open Access

Glossary

Open Access

Glossary

Some of the terms or abbreviations used in this manual may be unfamiliar to you. This glossary explains some of the more important ones.

ADP The *Angel Debug Protocol* is the communication protocol used between the ARM debugger and the target-resident Angel Debug Monitor, or the EmbeddedICE interface unit with ADP-compliant software (version 2.x).

ADW The *ARM Debugger for Windows* is the Windows debugger in the ARM Software Development Toolkit.

AMBA The *Advanced Microcontroller Bus Architecture* is a specification of a standard on-chip bus architecture, including the ASB and APB.

APB The *Advanced Peripheral Bus* is a bus specification for simple interfacing to low/medium speed peripherals for low power consumption. It is part of AMBA.

ARM7TDMI The ARM7TDMI test chip is an example of an ARM processor macrocell that is suitable for use on the ARM Development Board. Refer to the *ARM7TDMI Data Sheet* (ARM DDI 0029) for more information.

ASB The *Advanced System Bus* is a pipelined bus specification for multiple processors and high speed peripherals. It is part of AMBA.

ASIC The *Application-Specific Integrated Circuit* is used in this guide to mean an ARM processor core combined with application-specific peripherals on a single IC.

CPLD A *complex programmable logic device* is usually a collection of PAL type devices in a single package. The AMD MACH device is an example of a CPLD.

EmbeddedICE This is a term used to describe the additional hardware that is provided by debuggable ARM processors to aid debugging. The EmbeddedICE macrocell is fully described in the *ARM7TDMI Data Sheet* (ARM DDI 0029). The EmbeddedICE macrocell is controlled via the JTAG test access port, using an EmbeddedICE interface. This is an extra piece of hardware that allows software tools to debug code running on a target processor.

FPGA A *field programmable gate array*, or *logic cell array* (*LCA*) as it is sometimes called, is a type of *programmable logic device* (*PLD*). The ARM Development Board is fitted with one FPGA manufactured by Xilinx. The functionality of this device can be changed by the user if the appropriate design tools are available. Xilinx sells an appropriate tool set that can be interfaced to a

Open Access

variety of front end systems. These front end systems may be based on schematics or hardware description languages such as VHDL.

ICE An *in-circuit emulator* is a device that aids debugging of hardware and software. ARM debuggable processors such as the ARM7TDMI have extra hardware called EmbeddedICE to assist this process.

JTAG *Joint Test Action Group*. This acronym is usually used to describe the serial-like test port provided on many large silicon chips such as the ARM7TDMI.

LCA A *logic cell array* is a type of programmable logic device (PLD) also known as a field programmable gate array (FPGA).

MACH A MACH device is a example of a complex programmable logic device (CPLD). The ARM Development Board uses a number of MACH210 and MACH230 devices. Based on electrically erasable (EE) technology, they are reprogrammable. Using appropriate software (such as PALASM), the function of these devices may be changed by reprogramming in a standard programmer.

NISA *NISA* (*not-ISA*) is a name invented by ARM to describe the bus that connects the Advanced System Bus (ASB) to some standard peripheral devices such as the serial/parallel ports and PC card controller. It is a subset of the Industry Standard Architecture (ISA) bus found in most IBM compatible PCs.

PAL A *programmable array logic* device is a example of a programmable logic device (PLD). The PAL used on the ARM Development Board is a PALCE22V10. This has up to 22 inputs, ten outputs and ten programmable macrocells. Based on electrically erasable (EE) technology, it is reprogrammable. Using appropriate software (such as PALASM), the function of this device may be changed by reprogramming in a standard programmer.

PCMCIA The *Personal Computer Memory Card Association* produces a specification that details an interface suitable for connecting small credit card sized boards to larger host systems. The name PCMCIA is generally used to describe these cards, but its use has been superseded by the term PC card.

PLD A *programmable logic device*. See also PAL and FPGA.

PALASM A *programmable array logic assembler* is a low-cost, proprietary logic description language produced by Advanced Micro Devices (AMD) for their range of PLDs and CPLDs. It has been extensively used in the design of the ARM Development Board.

PLL A *phase locked loop* usually comprises a voltage controller oscillator, programmable divider, frequency comparator, and an integrator. These components allow programmable frequency

Open Access

clock to be generated which is locked to and stabilised by a reference clock input.

On the ARM Development Board, a single component performs this function. A reference crystal at 14.318MHz is used, and with three programmable inputs the device is able to generate 8 output frequencies from about 4–50MHz.

RDP The *Remote Debug Protocol* is the old communication protocol used between the ARM debugger and the target-resident Demon Debug Monitor or the EmbeddedICE interface unit with RDP-compliant software (version 1.x).

RPS The *Reference Peripheral Specification* is a specification of core peripherals to provide a standard ARM processor programmer's model. Peripherals specified are: interrupt controller, two timers, and a remap and pause controller.

SWI A *software interrupt* instruction generating an exception that causes the processor to call a handler to service the interrupt.

TAP The *Test Access Port* is part of the EmbeddedICE on-chip macrocell that interfaces between the JTAG pins and the scan chains on-chip..

VHDL VHDL is a hardware description language suitable for simulation and synthesis of logic circuits. The design for the FPGA on the ARM Development Board was completed using VHDL and synthesis tools from Compass. Xilinx tools were used to place and route the design.

Open Access

Index

A

ADP over JTAG 2-29
AMBA 4-2, 5-13, 6-2
Angel 2-6, 2-7
  and Demon 2-8
  comparison with EmbeddedICE 2-8
  configuring armsd 2-22, 2-25, 2-27
  configuring the Windows debugger 2-21, 2-24, 2-27
  Debug Protocol 2-7
  endianness 2-32
  Ethernet connection 2-8, 2-25
  Flash downloading 2-31
  on the ARM Development Board 2-8
  serial and parallel connection 2-22
  serial-only connection 2-21
  setting up ARM Development Board 2-20
  using EmbeddedICE 2-29
ARM Debugger for Windows 2-6
  configuring for Angel 2-21, 2-24, 2-27
  configuring for EmbeddedICE 2-15
  leds example 3-4
ARM Development Board
  architecture 5-3
  daughter board 6-2
  memory map 5-7
  mother board 6-2
  parallel port adaptor 6-2
  PC Card interface 5-27
  power supply 2-11
  powering up 2-11
  RAM region 5-9
  ROM region 5-12
  serial and parallel ports 5-25
  setting up 2-11
  setting up with Angel 2-20
ARM Software Development Toolkit
  versions 2-2
armsd 2-6
  configuring for Angel 2-22, 2-25, 2-27
  configuring for EmbeddedICE 2-18

Open Access

ARMulator 2-4
ATMEL devices 7-24, 9-3

B

BIGEND link 2-33, 6-12
Big-endian operation 5-6, 6-12

C

C source files
  for example code 7-2, 7-3
Centronics printer port 5-25
Code
  density 8-6
  performance 8-7
Counter/timer. *See* Timer

D

Daughter board 6-2
Debug Comms Channel 2-6, 2-9, 2-29
Debug extensions to ARM core 2-5
Debug monitor. *See* Angel
Debugger
  See ARM Debugger for Windows
Debugging systems 2-4
Demon
  and Angel 2-8
Dhrystone benchmarking test function 8-2
Diagnostic utilities
  diagnose 7-3, 9-4
  fl_wrt.c 7-24
  selftest 9-3
DIRN link 6-10
DRAM 5-10
  SIMMS 5-11

E

EmbeddedICE 2-4
  comparison with debug monitor 2-8
  configuring armsd 2-18
  configuring the Windows debugger 2-15
  configuring to match target core 2-16, 2-18
  connecting up 2-13
  debug extensions 2-5
  effect on debuggee 2-6
  endianness 2-18
  interface 2-5
  macrocell 2-5
  used by Angel 2-29
Endianness
  using Angel 2-32
  with EmbeddedICE 2-18
EPROM
  memory 6-13
EPROM SELECT link 2-20, 6-7, 9-3
EPROM/Flash 5-12
  configuring 6-7
Ethernet
  connecting Angel 2-8, 2-25
Ethernet Adaptor Kit 2-25
Example code 7-2
  C source files 7-2
  include files 7-3
  interrupt handler 7-9
  interrupt.c 7-9
  leds.c 7-4
  pcmc.c 7-12
  PCMCIA access 7-12
  PCMCIA memory card 7-14
  pcmcsram.c 7-14
  ROM download and operation 7-18
  serial.c 7-6
  worked example 3-2

Open Access

F

fl_wrt.c 7-24
Flash 5-12
  configuring 6-7
Flash downloader
  command-line operation 7-27
Flash downloading 2-26, 7-24
  Angel 2-31
  Windows operation 7-27
  with ARM Debugger for Windows 2-31
  with armsd 2-32
FPGA OK LED 2-12, 5-13
Frequency setting 6-4
Fusion stack 2-26

I

I/O peripherals 5-5
In-circuit emulator 4-2
Include source files
  for example code 7-3
INT TYPE link 6-10
Interrupt controller 5-14
  memory map 5-15
  register descriptions 5-16
INTERRUPT ENABLE link 2-23
Interrupt handler 7-10
  example code 7-9
interrupt.c 7-9
Inverse Assembler 4-2
IRQ_CLOCK 8-4

J

JTAG 2-14, 2-29, 2-32
  protocol conversion unit 2-5

L

LED port 6-9
LEDs 3-8
  diagnostic 9-5
  FPGA OK 2-12, 5-13
  green 2-12
  on EmbeddedICE unit 2-14
  on parallel port adaptor 2-25
  on power up 2-12
  PCA and PCB 5-28
  yellow 5-28
leds example 3-2
  ARM Debugger for Windows 3-4
  ARM Project Manager 3-3
  opening project files 3-3
  Thumb code 3-7
  using the command line 3-7
leds.c 7-4
Links
  BIGEND 2-33, 6-12
  DIRN 6-10
  EPROM SELECT 2-20, 6-7, 9-3
  INT TYPE 6-10
  INTERRUPT ENABLE 2-23
  REMAP 2-14, 2-20, 2-26, 2-32, 5-10, 5-12
  SELECT EPROM 2-33
  USETIC 4-3

M

Memory
  EPROM 6-13
Memory maps
  interrupt controller 5-15
  main 5-7
  RAM 5-9
  remap and pause 5-22
  remapping 6-5
  ROM 5-12
  timers 5-20

Open Access

Mother board 6-2

N

NISA address map 5-24
Normal (remapped) configuration 5-7

P

Parallel port 7-4
  configuring 6-9
  setting up 2-22
Parallel port adaptor 2-25, 6-2
PC Card 2-8, 2-11, 5-5, 5-27, 7-12, 7-14
PC Card Interface 5-27
PCA LED 5-28, 9-5
PCB LED 5-28, 9-5
pcmc.c 7-12
PCMCIA 2-25, 2-26, 5-5, 5-24, 5-27, 6-9, 7-12, 7-14
pcmcsram.c 7-14
Power supply
  for ARM Development Board 2-11
  for PC Cards 5-27
Powering up
  ARM Development Board 2-11
Project files 3-3
  opening 3-3
Protecting your card 2-11

R

RAM region 5-9
  DRAM 5-10
  SRAM 5-10, 6-6
  SSRAM 5-10
Real-time trace 4-2
Reference peripherals 5-13
  interrupt controller 5-14
  timers 5-18
Register descriptions
  interrupt controller 5-16
  timers 5-20
Remap and pause
  Clear Reset Memory Map 5-23
  Identification register 5-22
  memory map 5-22
  Pause control 5-22
  Reset Status register 5-23
REMAP link 2-14, 2-20, 2-26, 2-32, 5-10, 5-12
Reset configuration 5-7
ROM region 5-12
  EPROM/Flash 5-12

S

SELECT EPROM link 2-33
selftest utility 7-3, 9-3
Serial port 7-6
serial.c 7-6
SRAM 5-10
  speed 6-6
  width 6-6
SSRAM 5-10
System clock
  setting the frequency 6-4
System crash 3-2

T

Test interface 4-3
Testing the target system 3-3
Thumb
  benchmarking 8-6
  compiling leds example 3-7
Timers 5-18
  interrupt handler 7-10
  memory map 5-20
  register descriptions 5-20

Open Access

U

USETIC link 4-3

V

Vadem devices 5-27
Vadem VG-468 PCMCIA Interface
Controller 5-5
Vadem-ARM address bus connection 7-17
Vadem-to-ARM read modification 7-16

X

Xilinx FPGA 5-13

Open Access